

A Systematic and Semi-Automatic Safety-Based Test Case Generation Approach Based on Systems-Theoretic Process Analysis

Asim Abdulkhaleq, Institute of Software Technology, University of Stuttgart, Germany
 Stefan Wagner, Institute of Software Technology, University of Stuttgart, Germany

Software safety is a crucial aspect during the development of modern safety-critical systems. Software is becoming responsible for most of the critical functions of systems. Therefore, the software components in the systems need to be tested extensively against their safety requirements to ensure a high level of system safety. However, performing testing exhaustively to test all software behaviours is impossible. Numerous testing approaches exist. However, they do not directly concern the information derived during the safety analysis. STPA (Systems-Theoretic Process Analysis) is a unique safety analysis approach based on system and control theory, and was developed to identify unsafe scenarios of a complex system including software. In this paper, we present a systematic and semi-automatic testing approach based on STPA to generate test cases from the STPA safety analysis results to help software and safety engineers to recognize and reduce the associated software risks. We also provide an open-source safety-based testing tool called *STPA TCGenerator* to support the proposed approach. We illustrate the proposed approach with a prototype of a software of the Adaptive Cruise Control System (ACC) with a stop-and-go function with a Lego-Mindstorms EV3 robot.

CCS Concepts: •**Software and its engineering** → **Formal software verification**; **Software safety**; **Software testing and debugging**;

Additional Key Words and Phrases: safety-critical software, risk-based testing, STPA safety analysis, software safety, formal software verification, test case generation

1. INTRODUCTION

Software has become an indispensable part of many modern systems and often performs the main safety-critical functions. Hence, software safety must be analysed in a system context to gain a comprehensive understanding of the roles of software and to identify the software-related risks that can cause hazards in the system. Software safety as stated in [Alberico et al. 1999] is practically concerned with the software causal factors that are linked to individual hazards and ensured that the mitigation of each causal factor is traced from software requirements to design, implementation, and test. A software failure may lead to catastrophic results such as injury or loss of human life, damaged property or environmental disturbances. Therefore, it becomes essential to test the software components for unexpected behaviour before using them in practice [Minister of Defence 1991]. The Toyota Prius, the General Motors airbag and the loss of the Mars Polar Lander (MPL) mission [JPL 2000] are well-known software problems in which the software played an important role in the loss, although the software had been successfully verified against all functional requirements.

Software testing is a crucial process to assess the quality of the software and determine whether it meets its specified requirements. The term software safety testing [NASA-GB- 8719.13 2004] was introduced and implies that software testing should not only address functional requirements, but the software safety requirements as well. Therefore, the process for testing safety-critical software combines conventional testing and safety analysis approaches to focus the testing efforts in a specific way to address the safety of the software and test the critical risky situations. Fault Tree Analysis (FTA) [Vesely et al. 1981] and Failure Mode, Effects and Criticality Analy-

Author's addresses: A. Abdulkhaleq and, S. Wagner, Institute of Software Technology, University of Stuttgart, Universitätsstrae 38, 70569 Stuttgart, Germany
 YYYY 0000-0000/YYYY/01-ARTA \$15.00
 DOI: <http://dx.doi.org/10.1145/0000000.0000000>

sis (FMECA) [International 1967] are the approaches commonly used for the purpose of safety-based testing. However, these approaches focus only on single component failures and they have limitations to cope with complex systems including software. Leveson [Leveson 2011] noted that the primary safety problem in software-intensive systems is not software “failure” but the lack of appropriate constraints on software behaviour. The solution is to identify the required constraints and enforce them in the software and overall system design. Therefore, a new safety analysis technique called STPA [Leveson 2011] has been developed to overcome the limitations of the traditional techniques in addressing the unsafe scenarios of complex systems.

1.1. Problem Statement

The complexity of safety-critical software makes exhaustive software testing impossible. Therefore, we need to make sure that safety is sufficiently considered. Yet, many existing testing approaches and tools do not incorporate information from safety analysis. In case they do, they rely on traditional safety analysis techniques such as FTA and FMECA which focus on component failures instead of component interactions. A software safety testing approach integrated with alternative systems-theoretic safety analysis approaches such as STPA has been missing.

1.2. Research Objective

Our overall goal is to help software developers to more easily and effectively test safety-critical systems. In this paper, we focus on filling the aforementioned gap by a method which integrates generating safety-based test cases with the information derived during an STPA safety analysis in a systematic and semi-automatic way.

1.3. Contributions

To reach our research objective, we provide five main contributions: (1) We explore how to apply STPA to safety-critical software. (2) We provide an algorithm based on STPA to derive unsafe software scenarios and automatically translate them into a formal specification in LTL (Linear Temporal Logic) [Pnueli 1977] including timing. (3) We make use of specific STPA features (e.g. process model and the STPA safety requirements) to systematically derive a safe behavioural model. (4) We provide an algorithm to extract a safe test model from the safe behavioural model and check its correctness by automatically transforming it into an SMV representation (Symbolic Model Verifier) [McMillan 1993] and verify it against the STPA safety requirements using the NuSMV model checker [Cimatti et al. 1999]. (5) We propose an algorithm to automatically generate the traceability matrix between the STPA software safety requirements and the test model and generate the safety-based test cases for each safety requirement from the test model.

1.4. Terminology

We define the most relevant terms in table I to ensure a consistent terminology in this paper.

2. BACKGROUND

2.1. STPA Safety Analysis & Software Safety

STPA (Systems-Theoretic Processes Analysis) [Leveson 2011] is a top-down process based on the accident model called STAMP (Systems-Theoretic Accident Model and Processes). STPA is developed for generating detailed safety requirements of complex and modern systems to prevent the occurrence of unsafe scenarios in the systems. In STPA, the system is seen as a set of interrelated components which interact with

Table I: Terminology

| Terminology | Definition |
|--------------------------------|---|
| Software Safety | is the discipline of software assurance that is a systematic approach to identifying, analyzing, tracking, mitigating, and controlling software hazards and hazardous functions (data and commands) to ensure safe operation within a system [NASA-GB- 8719.13 2004]. |
| Accident | Accident (Loss) results from inadequate enforcement of the behavioural safety constraints on the process [Leveson 2011]. |
| Hazard | Hazard is a system state or set of conditions that, together with a particular set of worst-case environmental conditions, will lead to an accident [Leveson 2011]. |
| Unsafe Control Actions | The hazardous scenarios which might occur in the system due to provided or not provided control action when required [Leveson 2011]. |
| Safety Constraints | The safety constraints are the safeguards which prevent the system from leading to losses (accidents) [Leveson 2011]. |
| Process model | The process model is a model required to determine the environmental and system variables and states that affect the safety of the control actions and it is updated through various forms of feedback. [Leveson 2011] [Thomas 2013a]. |
| Process model variables | The process model variables are the safety-critical variables of the controller in the control structure diagram which have an effect on the safety of issuing the control actions [Thomas et al. 2012]. |
| Causal Factors | Causal factors are the accident scenarios that explain how unsafe control actions might occur and how safe control actions might not be followed or executed [Leveson 2011] [Thomas et al. 2012]. |
| Safe Behavioural Model | The safe behavioural model is a statechart notation that models the process model of a software controller in the STPA control structure diagram and and it is constrained by the STPA-generated software safety requirements (transition conditions). |
| Safe Test Model | The safe test model is an extended finite state machine model which is automatically constructed from the safe behavioural model. |
| Safety-based Test Cases | The safety-based test cases are set of the test cases which are generated from information derived during the safety analysis process. |

each other to provide a dynamic equilibrium through feedback loops of information and control. STPA has the following main steps: (1) Establish the fundamentals of the analysis (e.g. system-level accidents and the associated hazards) and draw the control structure diagram of the system (shown in Fig. 1). (2) STPA Step 1: Use the control structure diagram to identify the potential unsafe control actions. (3) STPA Step 2: Determine how each potentially unsafe control action could occur by identifying the process model and its variables for each controller and analysing each path in the control structure diagram.

The basic components in STPA are safety constraints, unsafe control actions, unsafe scenarios, control structure diagram and process models. A control structure diagram is made up of basic feedback control loops. An example is shown in Fig. 1. When put together, they can be used to model the high-level control structure of a particular system.

An extended approach to STPA is proposed by Thomas [Thomas 2013b] for identifying the unsafe control actions which are identified in STPA Step 1 based on the combinations of process model variables (context tables) of each controller in the control structure diagram. It also aims at performing STPA Step 2 based on a set of unsafe control actions which are identified in of STPA Step 1. Thomas [Thomas 2013b] mathematically discussed the formalization of STPA which can be used not only to identify

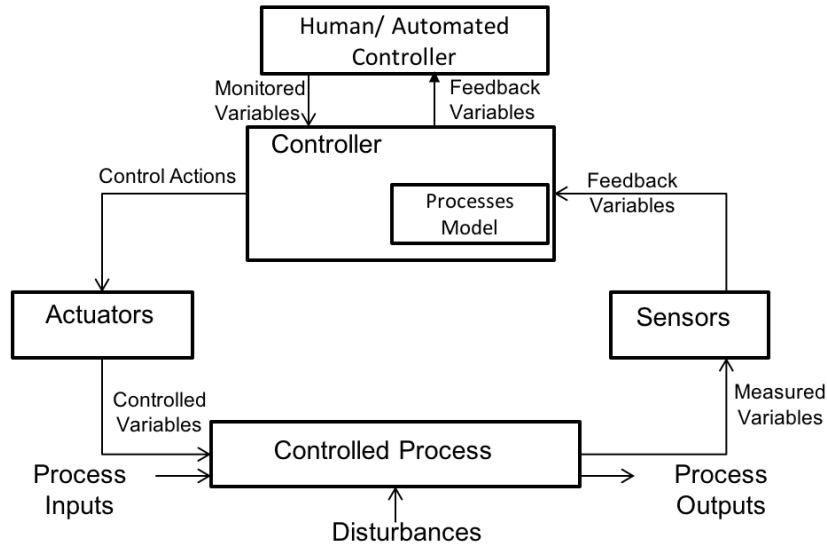


Fig. 1: A general feedback control structure of a software system

unsafe control actions and other control flaws, but also to generate model-based requirements that will enforce safe behaviours.

Definition 2.1 (A Control Structure Diagram). The Control Structure Diagram (CSD) of a software system can be expressed with five-tuples (CO, AC, SO, CP, CA) , where CO is a set (one or more) of the software controllers which control the controlled processes (CP) by issuing control actions to the actuators, AC is a set of the actuators which implement the control actions (CA) of the controller, CP is a set of the controlled processes which are controlled by controllers (CO). SO is a set of sensors which send the feedback about the status of the controlled process.

Each controller in the control structure diagram must contain a model of the assumed state of the controlled process, called the process model [Leveson 2011]. A process model contains one or more variables, the required relationships among the variables, the current state and the logic of how the process can change state. This model is used to determine what control actions are needed. It is updated through various forms of feedback [Leveson 2011]. The process model is a part of the internal state of the controller in the control structure diagram.

Definition 2.2 (A Software Controller). A software controller CO_i can be expressed formally as a two-tuple $CO_i = (CA, PM)$, where CA is set of the control actions and PM is the process model of the controller which has a set of process model variables (PMV), which are a set of critical variables P and states S that have an effect on the safety of CA : $P = \bigcup (P_1 = v_1 \dots P_n = v_n)$, where P_1 and P_n are process model variables of the software controller CO_i with their values v_1 and v_n .

In [Abdulkhaleq et al. 2015], we classified the process model variables of the software controller that affect the safety of the critical control actions into three types: 1) *Internal variables* which change the status of the software controller, 2) *Interaction interface variables* which receive and store the data/command/feedback from the other components in the system, and 3) *Environmental variables* of the environmental components that interact with or are controlled by the software controller.

2.2. STPA SwISs Approach for Software- Intensive Systems

Our preliminary algorithm [Abdulkhaleq et al. 2015] for deriving test cases from STPA results relied on using an existing model-based testing tool called ModelJUnit to drive the test cases. This algorithm is effective in deriving test cases but it has some limitations: 1) ModelJUnit requires that a behavioural model be written as a Java class, which represents the finite state machine of the system; 2) The ModelJUnit tool has not been developed with the purpose of safety-based testing and deriving the test cases from the safety analysis results; 3) there is no way to verify and check the correctness of a test input model of ModelJUnit against the safety analysis results; and 4) The ModelJUnit tool does not provide a traceability matrix between the safety requirements and the generated test cases.

¹<http://www.xstampp.de>

ronment (PDE) and Rich Client Platform (RCP). XSTAMPP supports performing the three main steps of STPA and provides an internal representation in XML for each STPA component to support possible future integration with other tools. STAMPP also supports the application of the *STPA SwISs* approach in identifying the unsafe scenarios and automatically deriving the corresponding software safety requirements. It also supports the formal verification activities by automatically expressing the refined software safety requirements into formal specifications in LTL. As a new extension to XSTAMPP, we developed an Eclipse plugin called STPA verifier² to automatically verify the refined software safety requirements which are expressed in LTL by using the model checkers (NuSMV and SPIN) in XSTAMPP.

In this paper, we discussed the algorithms to automate some activities of the *STPA SwISs* approach which are implemented in XSTAMPP to help the software and safety engineers in identifying the unsafe scenarios, automatically deriving the corresponding software safety requirements, and automatically expressing the refined software safety requirements into formal specifications in LTL.

2.3. Software Safety Testing

Software testing is one of the most important phases during the software development process to detect inconsistencies between the software implementation and its requirements. A popular testing approach called Model-based Testing (MBT) [Dalal et al. 1999; Apfelbaum and Doyle 1997] aims at automatically generating test cases using models extracted from software requirements. The model-based testing process involves creating a suitable model of the software's behaviour based on requirements or an existing specification to generate the test cases.

A big challenge in software testing is the design of test cases. To generate test cases, the tester needs first to understand the system specification and requirements. After that, the tester has to manually write test cases or automatically generate test cases from a model by using model-based testing tools. Automated generation of test cases involves that the system behaviour should be modelled in a suitable model. Over the years, there are many of the automated model-based test case generation approaches which have been developed by different techniques such as random generation algorithms [Prowell 2003], graph search algorithms [Kuan 1962; Broy et al. 2005], model-checking [Offutt et al. 2003], symbolic execution [Pretschner 2001] or theorem proving [Castanet and Rouillard 2002].

Software safety testing [NASA-GB- 8719.13 2004; Lutz 2000] is a crucial process in developing safety-critical systems to verify whether a software system meets its safety requirements. Safety-critical software should be tested extensively to ensure that the potential software-related hazards have been eliminated or controlled to a low level of risk.

A number of software behaviour models are in use today, several make good models for testing such as control flow charts [Harel 1987], finite state machines [Mealy 1955; Gill 1962], SpecTRM-RL [Leveson 2000], and sequence event diagrams [UML 2004]. A software behaviour can be described as an input sequence, actions, guards and output logic, or the data flow through the software modules and routines. In the following, we describe popular software behaviour models which are used to model software behaviour and generate test cases from these models:

Finite state machines are commonly used in software behaviour modelling and testing to generate test cases [Apfelbaum and Doyle 1997]. The finite state model (shown in Fig. 3) includes a set of states, a set of input events and the transition between them.

²<http://www.xstampp.de/STPAVerifier.html>

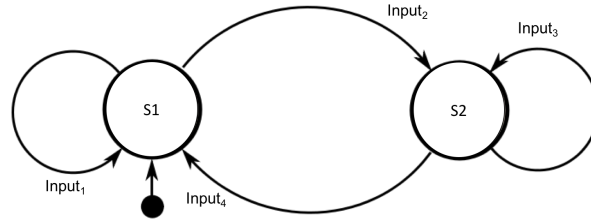


Fig. 3: The finite state machine model

Definition 2.3 (Finite State Machine (FSM)).

Let f be a finite state machine which can be defined with a 5-tuple [Mealy 1955]:

$f = (S, s_1, I, O, T_s)$, where S is a finite nonempty set of states with s_1 as the initial state, I and O are finite input and output alphabets, and T_s is a behaviour relation which defines all possible transitions of the finite state machine model.

Another software behaviour model is an Extended Finite State Machine (EFSM) which is an extension of the classical (Mealy) Finite State Machine (FSM) model with input and output parameters, context variables, operation and guards defined over context variables and input parameters. EFSM [Cheng and Krishnakumar 1993] is a common and very useful diagram to model the system behaviour and suitable for driving the test cases. EFSM contains nodes which represent the states of the system and the directed arcs which represent the transitions of the system from one state to another [Utting and Legeard 2007].

Definition 2.4 (Extended Finite State Machine (EFSM)).

Let M be an extended finite state machine which can be defined by the 6-tuple [Cheng and Krishnakumar 1993]:

$M = (Q, \sum_1, \sum_2, I, \vee, \wedge)$, where Q is a finite set of states, \sum_1 is a finite set of events, $I \subset Q$ is the set of initial states, \vee is the set of state variables, and \wedge is a finite set of transitions.

Statecharts [Harel 1987] were developed as a broad extension of the conventional formalism of finite state machines with notations of hierarchy, concurrency, and communication for describing the behaviour of complex or reactive software systems.

Definition 2.5 (A Statechart). Let SC be a statechart which can be defined with a 7-tuple [Harel 1987]:

$SC = (S, s_1, \lambda, \xi, \chi, \Omega, \Sigma)$, where S is a finite set of superstates, $s_1 \in S$ is as the start superstate which is either a state or a state-chart, λ is a transition function that maps the set of states S , ξ is a superstate function that maps the set of superstates onto itself, χ is an event function that maps the set of transitions T , $S \times S$ to a set of events, Ω is a finite set of events, and Σ is a default transition function that maps the set of states S to their default sub-state if it exists, or itself.

In [Harel 1987] Harel defined the statecharts language and the semantics of statecharts for complex systems. Simply, each Stateflow has a chart which is an independent state machine. Each chart has one or more states which are linked together by arcs labeled with transition information. The states can be also hierarchical states and contain a number of sub-states (children). Each state should have a type of state decomposition *OR STATE* or *AND STATE*. The *OR STATE* decomposition allows only

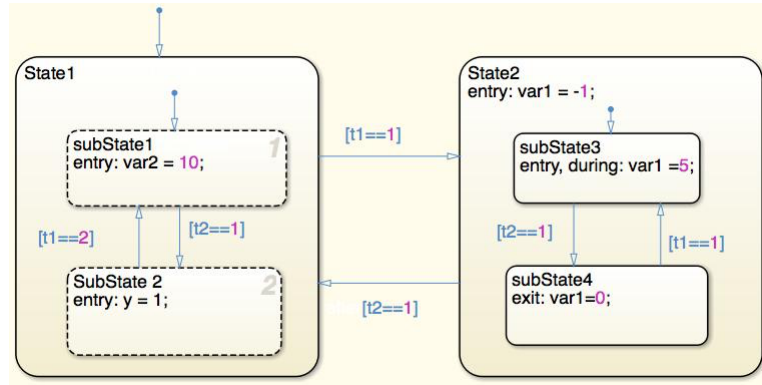


Fig. 4: The statechart model

one sub-state (which has a default transition) to be active at a time when the parent (superstate) is active whereas the *AND_STATE* decomposition allows all sub-states to be active when the parent (superstate) is active.

Based on Harel's statechart notations, the Simulink Stateflow model was developed by Mathworks³ to model event-driven (reactive) systems with enabling the representation of hierarchical state machine diagram, parallelism and history within statechart diagrams. The Simulink Stateflow is generally used to model the discrete controller in the model of a hybrid system where the continuous dynamics such as the behaviour of the plant and environment are specified using others capabilities of Simulink toolkit [Hamon and Rushby 2007a]. Recently, Matlab/Simulink has become a common model-based development tool for industrial software systems, which is widely used in various industries such as the aircraft, automotive, telecommunications, and transportation industries.

Figure 4 shows the semantics of the Simulink's Stateflow model. In Simulink's Stateflow, each state can be labeled with the following elements:

- name <of a state>
- entry: <entry actions are executed when a state is entered.>
- during: <during actions are executed when while in a state.>
- exit: <exit actions are executed when a state is left.>
- entry, during, exit: <combined actions in a state>

The Stateflow model in Fig. 4 includes two superstates (state 1 and state 2) with state decomposition *OR_STATE*. The superstate *State 1* has two sub-states with state decomposition *AND_STATE*. When the superstate *State 1* is active, then the both sub-states will be active in the same time. While the superstate *State 2* has two sub-states with state decomposition *OR_STATE*. That means when the superstate *State 2* is active, then only the sub-state with the default transition *subState 3* will be active.

The semantics of the Stateflow are defined informally in Simulink. Therefore, the Stateflow model need to be translated into a formal model supported by the verification approaches. Two different semantics for the Stateflow were proposed: 1) denotational [Hamon 2005] by Hamon and 2) formal operational [Hamon and Rushby 2007b] by Haman and Rushby. Therefore, it is important here to note that our proposed algorithm to translate the Stateflow model into an SMV model works only with a subset of the operational semantics of the Stateflow model. Moreover, our algorithm does not consider the semantic of join transitions which are allowed in the Stateflow semantics.

³<http://www.mathworks.com>

3. RELATED WORK

In the following, we will discuss the related work to our approach.

3.1. Risk-Based Software Testing

There are several software safety test techniques in the literature that combine safety analysis principles with model-based testing. Most of them use the term “Risk-based Testing”, which combines risk analysis approaches such as FTA, FMECA or Markov chains with software testing approaches (e.g. model-based testing) to create a prioritization criterion for generating test cases.

Redmill [Redmill 2004] explored the benefits of risk-based testing as the basis of test planning in the software testing process and how to understand the risks of the system to focus test efforts. He does not show how to generate the test cases from the risk analysis approach.

Zimmermann et al. [Zimmermann et al. 2009] proposed a refinement-based approach to the reliability analysis of safety-critical systems. They used statistical testing as a model-based testing technique and a Markov chain model to model the system under test. They also used FTA and FMECA as risk-based analysis techniques to identify the critical situations that represent high risk.

Kloos et al. [Kloos et al. 2011] proposed a model-based testing approach which uses the information derived from FTA in combination with a system model to generate the risk-based test cases. They used FTA to select, generate and prioritize the test cases. They derived the test cases from the combination of fault trees and a basic system behaviour model called the “base model”.

Our approach uses a similar idea of combining a risk analysis approaches with model-based testing. The main difference is that we employ STPA for safety analysis which is based on system and control theory rather than reliability theory like FTA and FMEA. STPA copes with the analysis of complex, modern systems and tackles the dynamic behaviour of the system by treating safety as a control problem. Furthermore, STPA provides an abstract model of the system under analysis called the safety control structure diagram which views all main interacting components including the software components of the system. This allows us to directly construct the test model from the control structure diagram and constrain its transitions with the STPA-generated safety requirements.

3.2. Translating Simulink Models into Models Supported by the Verification Approaches

A considerable amount of work has been done on translating Simulink models into models supported by formal verification approaches. In the following, we will discuss the most related work:

Banphawatthanarak et al. [Banphawatthanarak et al. 1999] developed a tool called *sf2smv* that generates input for the symbolic model checker SMV [McMillan 1993] from Stateflow models. In our work, we use the same concept for translating Simulink’s Stateflow into the SMV model. As *sf2smv* is not available yet, however, it is difficult to compare it with our approach in detail.

Meenakshi et al. [Meenakshi et al. 2006] discussed the principles of translating Simulink models into an input language of a suitable model checker and providing reverse translation of traces violating requirements into Simulink notation for playback. They developed a translator from Simulink to the model checker NuSMV [Cimatti et al. 2000]. The translator takes a Simulink model as input and generates an equivalent NuSMV model. However, this translator is restricted only to discrete Simulink models and support only the basic blocks of Simulink (e.g. logical block or Selector

block) that forms the finite state model of a system. Moreover, the translator does not support the translation of Simulink Stateflow into the input language of NuSMV.

Chen and Dong [Chen and Dong 2006] proposed a systematic approach to translate Simulink diagrams to Timed Interval Calculus (TIC) [Fidge et al. 1998], a notation extending Z to support real-time system specification and verification. The translated TIC specification covers the functional and timing aspects of the Simulink blocks. This work aims to guarantee the correctness of control systems. However, this work does not cover Stateflow diagrams.

Chen et al. [Chen et al. 2012] proposed an approach to systematically translate Stateflow diagrams to CSP# [Sun et al. 2009a]. They developed a translator which is integrated inside the PAT model checker [Sun et al. 2009b] to automate the process with support of different Stateflow features. This work aims to validate the functional correctness of Stateflow diagrams by detecting the bugs in the Stateflow model. The properties to be verified are declared in the CSP# model with preprocessor such as *#define*. The translation process preserves the execution semantics of Stateflow and considers advanced Stateflow modelling features such as implicit events and history junctions.

Ferrante et al. [Ferrante et al. 2012] developed a modified tool called Parallel NuSMV (PNuSMV) based on NuSMV model checker [Cimatti et al. 2000] that integrates the ManySAT parallel STA solver [Hamadi et al. 2008]. This tool is part of the formal specification verification framework for the formal verification of Simulink/Stateflow models. The tool translates a subset of Simulink blocks (e.g. logical operators and arithmetic blocks) into the NuSMV meta model. The interesting properties are expressed as temporal logic to be verified with the PNuSMV tool. However, this work does not consider translating the Stateflow model into the NuSMV specifications.

In very recent work, Yang et al. [Yang et al. 2016] presented a tool for the translation of Stateflow models to timed automata to check the correctness of the Stateflow model. The translated model is used as an input to Uppaal timed automata [Alur 1999]. They used Uppaal to analyse the timing behaviour of the system and check both safety and liveness properties of timed automata. We use a similar principle of transforming the Simulink's Stateflow model into an intermediate model with consideration of the state decomposition (AND_STATE and OR_STATE) and the attached actions (Entry, During and Exit). The main contribution of their work, however, is an approach to model check Simulink models which is not our main focus in this paper. Our contribution is to visualise the STPA process model, which is created during the safety analysis, with the Stateflow notation and check its correctness against the software safety requirements. In this way, we ensure that both models contain the same specifications (e.g. names of states, variables and control actions) before using it for generating test cases.

In conclusion, the existing work provide a great basis for our approach but are different in their focus. They concentrate on model checking Simulink models, not the integration with safety analysis or testing. To the best of our knowledge, there is no existing work on constructing the Stateflow diagram based on the information derived during a safety analysis for test case generation.

3.3. Generating Test Cases from Extended Finite State Machine

Over the years, many approaches have been developed to generate test cases from statechart diagrams. The idea behind these approaches is to transform the statechart diagram into an Extended Finite State Machine (EFSM) and generate the test cases from this model. In the following, we will discuss the most related work:

Ural [Ural 1987] proposed a method to transform the extended finite state machines into a flow graph and generate test sequences. This method is based on the principles

of using data flow analysis techniques in software reliability [Fosdick and Osterweil 1976] to trace the flow graph and generate test cases.

Bourhfir et al. [Bourhfir et al. 1997] proposed a unified method for automatic executable test case and test sequences generation which combines both control and data flow testing techniques with control flow criteria (Unique Input Output) and the all-du paths coverages criterion. Their approach generates only executable test cases for EFSM-specified systems by using symbolic evaluation techniques.

Kim et al. [Kim et al. 1999] proposed an approach to generate test cases from UML state diagrams based on the conventional control and data flow analysis. The authors have first transformed the state diagrams into EFSM with consideration of the hierarchical and concurrent structure of states (flattened and broadcast) of the UML state diagrams. Then, the EFSM are transformed into the flow graphs. They applied the conventional data flow analysis techniques to the resulting flow graph to generate the test cases. However, they focused only on identifying possible control and data flow and not the values of input variables.

Hong et al. [Hong et al. 2000] developed a method for the selection of test sequences from statecharts. The method is based on the STATEMATE semantics of statecharts by Harel [Harel and Naamad 1996]. The basic idea is to transform the statechart into an EFSM which contains all the possible runs of the statechart. The authors have considered the input variables in the EFSM which was generated from the state machine diagram. The resulting EFSM model will then be transformed into a flow graph to generate test sequences that cover all associations between definitions and uses of each variable that appear in the original state machine. The authors used the existing method of Ural [Ural 1987] to transform the EFSM into a flow graph that models the flow of both the control and the data in the statechart.

In conclusion, our approach uses a similar principle of generating test cases from the test model by using graph search algorithms (depth-first search, breadth-first search and both combined) which are presented in the aforementioned mentioned approaches. However, we choose different test coverages criterion to generate test cases such as all states coverage, all transition conditions coverage and the STPA software safety requirements coverage. However, our approach transforms each state in the safe test model as an executable Java script function that takes the state variables which are declared in the state actions (Entry, During, Exit) as parameters and execute their equations to update their values. The updated values of these variables will be used to check the transition condition and determine the next state. Moreover, our approach provides traceability between the software safety requirements and test model and traceability between the software safety requirements and the generated test cases.

3.4. Generating Test Cases from Simulink Models

Few research has concentrated on the subject of the automatic generation of test cases from Simulink Stateflow: Zhan and Clark [Zhan and Clark 2008] developed an approach for automatic testing of Matlab/Simulink models. Their approach is a search-based test data generation and selection approach. It uses the first search-based approach to generate test data.

Păsăranu et al. [Pasareanu et al. 2009] proposed a framework for model-based analysis and test case generation for flight software of a NASA flight mission based on the Simulink Stateflow and UML representations. They used Java path finder⁴ and symbolic path finder⁵ to generate test cases from both UML and Simulink/Stateflow models. The proposed framework is based on the concept of using model checking to

⁴<http://javapathfinder.sourceforge.net>

⁵<http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbolic>

generate test cases. The framework takes the models which are created by using different modelling environment and enables their analysis with model checking and test case generation approaches.

Windisch [Windisch 2009] proposed an automation approach for search-based testing of continuous functional models like Simulink Stateflow models. The method demonstrates how search-based testing techniques can be applied to a continuous functional model such as Simulink/Stateflow to generate test cases.

Li and Kumar [Li and Kumar 2012] proposed an automatic method for test data generation for Simulink Stateflow based on its translation to input/output extended finite automata model. The method involves that the Simulink Stateflow shall translate to an input/output extended finite automate model. Each path in Input/output extended finite automate model represents a computation sequence of the Simulink Stateflow diagram. They implemented the proposed method by using two model checking techniques and constraint solving. The NuSMV model checker is used to map the input/output of the extended finite automata to a finite abstracted transition system modeled in SMV and generate test cases by checking each path in the I/O of the extended finite automata against the resultant model.

We differentiate our work here from the aforementioned approaches by automatically generating the safe test model from the Stateflow model which is constructed from the safety analysis specification. Our approach also shows how to validate the correctness of the safe test model against the software safety requirements by using the NuSMV model checker.

4. THE PROPOSED APPROACH

In this section, we propose an automatic safety-based test case generation approach for deriving test cases directly from the STPA safety analysis results. The proposed approach follows the main steps of the STPA SwISs approach and provides a high degree of automation of each step.

We introduce a running example that we will use to illustrate our approach. It is the software controller of a train door system. Let us assume the software controller of the train door control system was designed to open and close a door of a train and monitor the status of the door. The controller works by receiving information about the position and the status of the door by a sensor. The door controller also receives input from external sensors about the position of the train and whether an emergency is happening. Then, the controller issues the door open or close commands. The actuator will receive these commands and apply mechanical force on the physical door.

Figure 5 shows the main steps of the approach which includes four steps: 1) deriving the software safety requirements of a software controller by following the STPA SwISs approach [Abdulkhaleq et al. 2015] and automatically expressing them in LTL, 2) constructing the safe behavioural model of the software controller with the state-chart notations in Simulink, 3) transforming the safe behavioural model into an input model of the NuSMV model checker and checking the correctness of the generated model against the STPA and safety requirements expressed in LTL; and 4) automatically generating a safety-based test model and deriving the safety-based test cases from this model.

In the following sections, we describe the four major activities of the proposed approach in more detail.

4.1. Deriving Software Safety Requirements

This step starts by applying STPA to the system specification to identify STPA software safety requirements and the potentially unsafe scenarios which the software can contribute to. The algorithm starts by establishing the fundamentals of analysis by de-

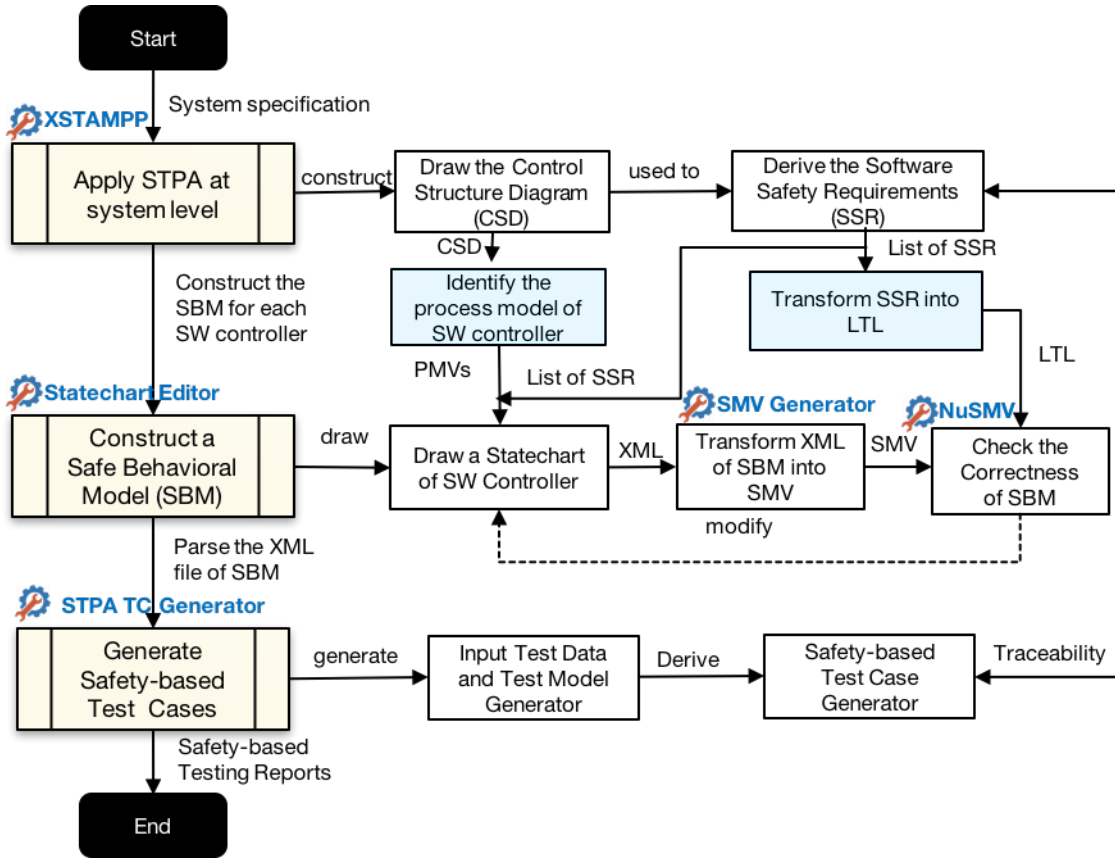


Fig. 5: An overview of the proposed approach

termining the system-level accidents (*ACC*) and the associated system-level hazards (*HA*) which the software can lead to or contribute in. Next, the algorithm demands that the safety control structure diagram of the system shall be constructed from the system specifications. The software here is the controller in the control structure diagram.

For our running example of the train door, we can reuse the STPA analysis in [Thomas and Leveson 2011]. For example, they came up with the accident *ACC.1: A person is injured while the train closed the door*. A system-level hazard that can lead to this accident is, for example, *HA.1: Door closed the door while a person is in the doorway*. The control structure diagram of the train door system is shown in Fig. 6). The control structure diagram includes: 1) software door controller; 2) door actuator; 3) physical door; 4) and door sensor.

For each software controller component in the control diagram, its software safety requirements can be derived by performing the following steps:

- (1) STPA Step 1: Identify unsafe control actions. In this step, the safety analyst will identify the potentially unsafe software control actions for each software component that can lead to one or more of the defined system hazard *HA*, as follows:
 - (a) Identify all safety-critical Control Actions (*CAs*) that can lead to one or more of the associated hazards (*HA*). For example, the software controller of the train

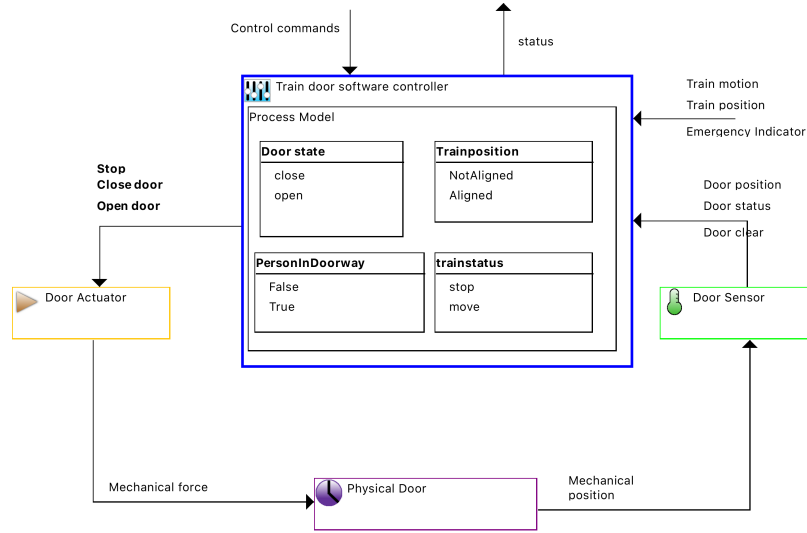


Fig. 6: The control structure diagram of train door system with the safety-critical process model variables

door system issues three control actions: *open door*, *close door*, *stop opening or closing door*.

- (b) Evaluate each CA with four general types of hazardous behaviours to identify the Unsafe Control Actions (UCAs): (a) a control action required for safety is not provided, (b) an unsafe action is provided, (c) a potentially safe control action is provided too early, too late or out of sequence and (d) a safe control action is stopped too soon or continued too long. For example, the control action *CA.1: close door* can be unsafe and lead to the hazard *H1*. It is then *UCA1.1: The software controller closes the door while a person is in the doorway*.
- (c) Translate the identified UCAs manually into informal textual Software Safety Requirements (SSR). For example, the corresponding software safety requirement of *UCA1.1* is *SSR1.1: The door software controller must not close the door while a person or object is in the doorway*.
- (d) Identify the process model and its variables and include them in the software controller in the control structure diagram to understand how each UCA could occur. The process model describes the states of the software controller (only critical states which are relevant to the safety of the control actions) and their variables describe the software communication, input and output. Figure 6 already shows the process model of the software controller. The process model has four process model variables *PMV*: 1 state variable *S* ($S_1 = \text{door state}$) and 3 variables *P* ($P_1 = \text{personInDoorway}$, $P_2 = \text{Train status}$ and $P_3 = \text{Trainposition}$)
- (e) Automatically generate the critical set of combinations of the process model variables for each control action (CA). Each combination should be evaluated within two contexts ($C_1 = \text{Providing CA}$ or $C_2 = \text{Not Providing CA}$) to determine whether the control action is hazardous in that context or not. A control action *CA* could be considered hazardous in context *C* if only a combination of process variables related to *CA* leads to a system-level hazard $H \in HA$. The context $C_1 = \text{Providing CA}$ has three types of sub-contexts: *context incorrectness*, in which the unsafe control action commanded incorrectly and caused a hazard (any time), *context real-time execution*, in which the unsafe control

action commanded in a wrong timing (too early or too late) or sequence, and *context execution mechanism*, in which the unsafe control action commanded in a wrong mechanism of execution (applied too long or stopped too soon). For example, the process model variables that have an effect on the safety of providing or not providing the control action *CAI: close door* are *door state, train position, person in door and train status*. An example of the critical set of combinations which can be generated based on the process model variable values for the *close door* control action in the context of providing C_1 is C_{s_1} : *door state = open, train position = aligned, a person in doorway = yes and train status = stop*

- (2) STPA Step 2: Identify the unsafe software scenarios for each unsafe control action. Based on the results of STPA Step 1, the safety analyst will identify the unsafe software scenarios for each unsafe control action *UCA* as follows:
 - (a) Identify the potentially unsafe critical combination of unsafe software control actions and evaluate it to identify the potential unsafe scenarios of the software controller that cause accidents. For example, an unsafe scenario which can be derived from the critical combination C_{S_1} is *SC1.1: The door software controller provided the control action close door while the train is stopped and train position is aligned, door state is open and a person is in the doorway*.
 - (b) Refine software safety constraints based on the unsafe scenarios of the software controller. For example, the software safety requirement which can be formulated from the unsafe scenario *SC1.1* is *SSR1.1: The door software controller must not provide the close door command while the train is stopped, the train position is aligned with the platform, the door state is open and a person is in the doorway*

Definition 4.1 (Refined Unsafe Control Action). The refined unsafe control action (*RUCA*) is a four-tuple (CA, Cs, C, TC) , where CA is a control action which causes a hazard $H \in HA$, $Cs = \bigcup(\mathcal{P}_1 = v_1, \dots, \mathcal{P}_n = v_n)$ which is a critical set of combinations of the relevant process model variables PMV of CA , C is a context where providing or not providing the control action CA is hazardous, and TC is the type of context **providing** of control action CA (**any time, too early or too late**).

To automatically translate each critical combination of process model variables for each control action CA into the unsafe software scenarios, we set the following rules:

Rule 1: Each refined unsafe control action (*RUCA*) in the context of **Providing** (C_1) of a control action CA_i can be expressed as:

$RUCA_i = \langle CA \rangle$ **provided** $\langle TC \rangle$ **is hazardous when** $\langle Cs = \bigcup(\mathcal{P}_1 = v_1, \dots, \mathcal{P}_n = v_n) \rangle$ occurred.

Rule 2: Each refined unsafe control action (*RUCA*) in the context of **Not Providing** (C_2) of a control action CA_i can be expressed as:

$RUCA_i = \langle CA \rangle$ **Not provided is hazardous when** $\langle Cs = \bigcup(\mathcal{P}_1 = v_1, \dots, \mathcal{P}_n = v_n) \rangle$ occurred.

By using the rules 1 and 2, we refine the unsafe control actions which are identified based on the combination set of process model variables. The software safety requirements are generated automatically from the refined unsafe control actions. Based on definition 3, we identify the following rules which are used to automatically generate

the Refined Software Safety Requirements ($RSSR$):

Rule 3: Each $RUCA_i$ in the context **Providing** (C_1) of control action CA_i can be transformed automatically into a new software safety requirement as follows:

$RSSR_i = \langle CA \rangle$ **must Not be Provided** $\langle TC \rangle$ **when** $\langle Cs = \bigcup (\mathcal{P}_1 = v_1, \dots, \mathcal{P}_n = v_n) \rangle$ occurred.

Rule 4: Each $RUCA_i$ in the context **Not Providing** (C_2) of control action CA_i can be transformed automatically into a new software safety requirement as follows:

$RSSR_i = \langle CA \rangle$ **must be Provided when** $\langle Cs = \bigcup (\mathcal{P}_1 = v_1, \dots, \mathcal{P}_n = v_n) \rangle$ occurred.

4.2. Automatically Formalizing Safety Requirements in Linear Temporal Logic (LTL)

In [Abdulkhaleq and Wagner 2015a], we described an algorithm to formalize the safety requirements in LTL. Here, we extend it to include software safety requirements that include timing. By using rules 3 and 4, each refined software safety requirement $RSSR_i$, which is identified from the refined unsafe control action $RUCA_i$, can be transformed automatically into a formal specification in LTL.

Rule 3 defines three types of software safety requirements, which means that the control action CA_i must not be provided in the type of context $TC =$ **any time**, **too early or too late** when the critical combination Cs_i of the relevant process model variable values occurred. Each type of software safety can be transformed automatically into formal specification by the following rules:

Rule 3.1: Each $RSSR_i$ derived from the context of providing control action CA_i **any time** (without delay) can be automatically transformed into LTL as:

$LTL_i = G (Cs_i \rightarrow ! (controlAction == CA_i)),$ where $Cs_i = \bigcup (\mathcal{P}_1 = v_1 \wedge \dots \mathcal{P}_n = v_n).$

Rule 3.1 means that it always (G) the software controller should not (!) provide a control action CA_i when the values of the critical combination Cs_i have been occurred.

Rule 3.2: Each $RSSR_i$ derived from the context of providing control action CA_i **too early** can be automatically transformed into LTL as:

$LTL_i = G (((controlAction == CA_i) \rightarrow Cs_i) \& ! ((controlAction == CA_i) U Cs_i)).$

Rule 3.2 means that a software controller should always (G) not provide control action CA_i before the occurrence of critical combinations set Cs_i still not become true in the execution path and that it well provides the CA_i when the combination of Cs_i holds.

Rule 3.3: Each $RSSR_i$ derived from the context of providing control action CA_i **too late** can be automatically transformed into LTL as:

$LTL_i = G ((Cs_i \rightarrow (controlAction == CA_i)) \& !(Cs_i U (controlAction == CA_i))).$

Rule 3.3 means that the software controller should always (G) not provide a control action CA_i too late while the occurrences of the critical set of combinations has become previously true in the execution path.

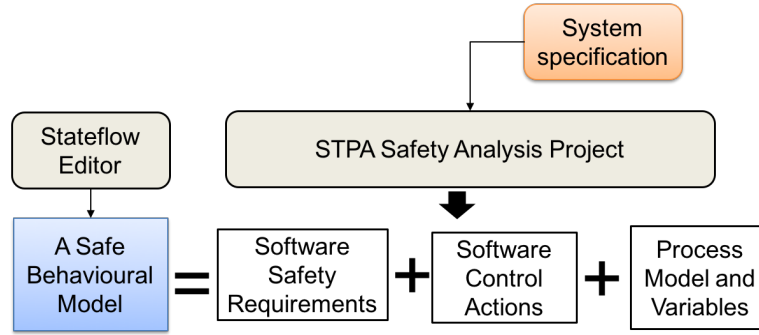


Fig. 7: A safe software behavioural model

Rule 4.1 defines one type of the software safety requirements which is the context of not providing a control action CA_i when it is required. This type can be expressed into LTL by the following rule:

Rule 4.1: Each $RSSR_i$ derived from the context of **Not providing** of control action CA_i can be automatically transformed into LTL as:

$LTL_i = G (Cs_i \rightarrow X(\text{controlAction} == CA_i))$, where $Cs_i = \bigcup (\mathcal{P}_1 = v_1 \wedge \dots \mathcal{P}_n = v_n)$.

This rule means that the occurrence of a critical set of combination values always implies that the software controller must provide the control action CA_i at the next time step (X) without any delay.

For example, the corresponding LTL of the software safety constraint $SSR1.1$ of the train door software controller can be specified as follows:

$LTL_{1.1} = G (((\text{trainstatus} == \text{stop}) \ \&\& \ (\text{doorstate} == \text{close}) \ \&\& \ (\text{trainposition} == \text{Aligned}) \ \&\& \ (\text{PersonIndoorway} == \text{TRUE})) \rightarrow !(\text{controlAction} == \text{closeddoor}))$.

4.3. Constructing a Safe Software Behavioural Model

To generate the safety-based test cases, the information derived from the STPA safety analysis must be integrated into a suitable model which should visualise the process model variables of each software controller and their relations in a control structure diagram. For this purpose, we select the Stateflow [MathWorks 2016] diagram notations to visualise the automation model of each software controller. The Stateflow diagram is a visual notation for describing dynamic behaviour, including the hierarchy, concurrency and communication information. The idea here is to build a model from STPA results with a modelling editor (e.g. Simulink) that supports the export of the statechart notations as XML specifications.

Definition 4.2 (Safe Behavioural Model (SBM)). Let SBM be a Safe Behavioural Model (shown in Fig. 7) which can be expressed by a three-tuple (PMV, T, CA) , where PMV is a set of the safety-critical process model variables: critical variables P and S states: $P \subset PMV$ and states $S \subset PMV$, T is the set of transition conditions which are extracted from the STPA refined software safety requirements $RSSR$ that are refined based on PMV , and CA is the set of the critical software control actions.

Each transition T_i of the safe behavioural model is expressed with the syntax $T_i = IE [SSR] / TA$, where IE is the input event that causes the transition T_i , SSR is a safety requirement which is a Boolean condition that constrains the transformation

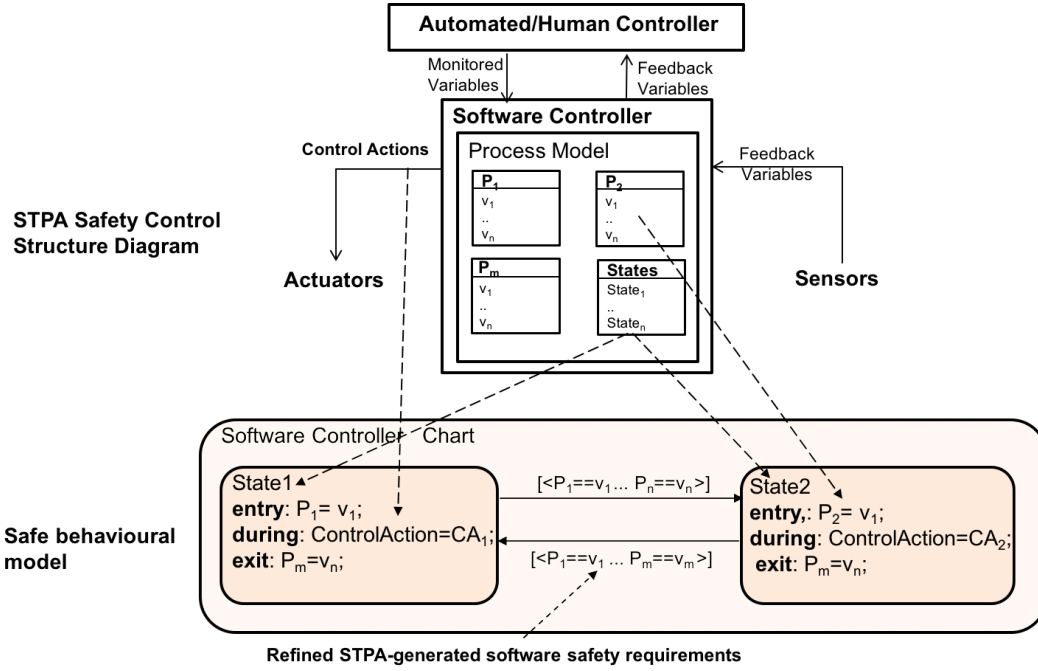


Fig. 8: Mapping the process model variables and control actions into the safe software behavioural model

from the current state to the next state, and TA is an action that will be executed when the Boolean expression is valid. Each state in the Stateflow model has three optional types of actions: *Entry*, *During* and *Exit* actions. Entry actions execute when the state is entered, *During* actions execute when the state is active, an event occurs and no valid transition to another state is available, and *Exit* actions execute when the state is active and a transition out of the state occurs [MathWorks 2016]. These actions are used to determine how to change the current state of the software controller to the next state.

The syntax of the Stateflow in Simulink allows to combine these three actions that execute the same tasks in a state. To change values of the process model variables $P = \bigcup (P_1 = v_1 \wedge \dots \wedge P_n = v_n)$ in a state, we used these actions of each state in the safe behavioural model to determine how each value of the process model variable (P_i) can be changed when the software controller enters or exists or this state. For example, the process model variable P_i in the process model of the software controller can be written in a state as an Entry, During or Exit action or combined state actions as follows: *entry, during, exit* : $P_i = <new\ value\ of\ P_i>$.

As the transition condition is derived from the refined STPA software safety constraints, the new value of each process model variable will be used to check the transition condition of the current state to determine what is the next state. We also used these state actions to determine which control action of the software controller can be dispatched on entering, during or exiting the current state. Figure 8 shows how to map the internal process model variables of the software controller and its control actions into the safe behavioural model. We identify the rules of constructing a safe software behavioural model from the STPA results as follows:

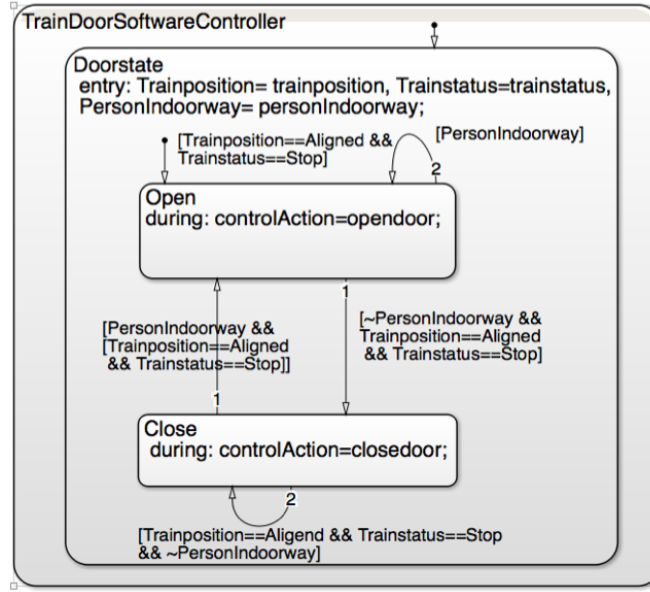


Fig. 9: The safe behavioural model of the train door software controller

- The process model of each software controller in the STPA control structure diagram will be visualised by a chart in the Stateflow model. For example, the Stateflow model of the door software controller (as shown in Fig. 9) has one chart which contains one superstate *door state*.
- The safe behavioural model should contain all internal states *S* and the safety-critical process variables *P* of the software controller in the STPA control structure diagram. As shown in figure 9, the door software controller contains one internal state *doorstate* which has two substates *close* and *open*.
- All process model variables of the software controller in the STPA control structure diagram should be declared in the safe behavioural model. As an example, we declared the three process model variables of the train door software controller as follows: *Trainposition* (Enum), *Trainstatus* (Enum) and the *personIndoorway* (Boolean)
- The safe behavioural model should constrain the transitions using the STPA software safety requirements (constraints) which are identified based on the rules 3 and 4. For example, the train door software controller must provide the control action *open door* when the train is stopped and aligned with platform. We used this information to constrain the default transition of the state *open*.
- Define an enumeration data type variable named *controlAction* in the safe behavioural model which takes all control actions of the software controller in the STPA control structure diagram as its value. For example, we defined the control actions of the train door software controller *open door*, *close door* and *stop* as enumeration data type in the safe behavioural model.
- The *controlAction* variable will be used as an entry/during/exit action of internal states of the safe behavioural model to show which control action will be issued when the software controller enters or exits a state.

4.4. Automatically Transforming a Safe Software Behavioural Model into an SMV Model

To check the correctness of the safe behavioural model and ensure that the safe behavioural model of the software controller satisfies all STPA software safety requirements, the safe behavioural model must be verified against the generated LTL formulae. For this purpose, we developed an algorithm that automatically transforms the SBM model created in the Simulink editor into an input language of a model checker such as SMV (Symbolic Model Verifier), automatically parses the LTL formulae from the STPA data model and includes them into an SMV model. To verify the SMV model against the STPA software safety requirements, we use the NuSMV model checker. In case that the SMV model does not satisfy a given LTL of a software safety requirement, the NuSMV model checker will return a counterexample. A counterexample contains information that shows why the given LTL formula of a software safety requirement is not satisfied. Based on the counterexample's information, the safe behavioural should be modified. As the LTL formula contains information about the state of software controller s_i and the control action CA , therefore, the modification of the safe behavioural model involves changes to the transition conditions or the initial values of the variables of the state s_i in which the model violated the given LTL formula. This step continues until the safe behavioural model satisfies all STPA-generated software safety requirements.

The algorithm of generating the SMV model is divided into three sub-algorithms: 1) *generate STPA data model* which parses XML specifications of the STPA project created in XSTAMPP (shown in algorithm 1); 2) *generate Stateflow (safe behavioural model) data model* which parses XML specifications of a Stateflow model and generate a tree of Stateflow states (TSf) in which a node represents one Stateflow state (shown in algorithm 2); and 3) *generate SMV model* which transforms the STPA data model and Stateflow data model into SMV specifications (shown in algorithm 3).

4.4.1. Parsing the STPA project created by XSTAMPP. Algorithm 1 shows the process of parsing the STPA project created by XSTAMPP. The algorithm process accepts the STPA project file F as input. Then, it parses the XML specification of the STPA project into the corresponding data model $DataModel$ which represent all data in an STPA project (see lines 1 – 3). For each software controller in the control structure diagram, a data model DM_{SW} will be created to store the information about the software controller such as its critical control actions, process model and its variables, software safety requirements and the generated LTL formulae (see lines 4 – 5). The algorithm will fetch the information of each software controller and store them in the corresponding lists (see lines 6 – 9) and add these lists into the data model of the software controller (see lines 11 – 14). The output of this algorithm is a list of the data model of the software controllers in the safety control structure diagram (see line 16).

4.4.2. Parsing the Stateflow model created by Simulink/Matlab. Algorithm 2 shows how to parse the XML specifications of the Stateflow model stored in a Simulink/Matlab file. The input of this algorithm is an XML file of the Simulink Stateflow file (Sf) which contains XML specifications of the Stateflow model.

The structure of the Stateflow model allows a multilevel hierarchy of states in which a state $S_{i,j}$ can contain sub-states with different types, where i indicates the number of the level hierarchy of the Stateflow model ($i = 0 \dots n$), j is the number of states, and n is the total number of levels in the Stateflow model. Therefore, the process of algorithm 2 traverses recursively the Stateflow data model based on the depth-first search algorithm to consider all sub-states of the superstate and add them to the tree of Stateflow. Each Stateflow model has two kinds of state decomposition: OR states (exclusive) and AND states (parallel) [MathWorks 2016]. The Stateflow semantics allow every state to

Algorithm 1 Generate STPA Data Model**Input:** F : A STPA project file**Data:** CAs = a list of control actions, $PMVs$ = a list of process model variables, $SSRs$ = a list of software safety requirements, and $LTLs$ = a list of generated LTL formulae of SSR . $DataModel$ = a data model which stores all information of STPA project F .**Output:** $DataModel_{SW}$ = a list of the data model of the software controller $CO \in F$.**Description:**

```

1: URL schemaFile ("/hazschema.xsd")
2: XSMModel LoadXMLSchema (schemaFile)
3: DataModel ParseXMLSchema ( $F$ )
4: for each  $SW_i$  Controller in  $DataModel$  do
5:   Create a new data model  $DM_{SW}$  for  $SW_i$  Controller.
6:   Fetch:  $CAs \leftarrow DataModel.fetchControlActions()$ 
7:   Fetch:  $PMVs \leftarrow DataModel.fetchProcessModelVariables()$ 
8:   Fetch:  $SSRs \leftarrow DataModel.fetchSoftwareSafetyRequirements()$ 
9:   Fetch:  $LTLs \leftarrow DataModel.fetchLTLs()$ 
10:  Add  $DM_{SW}.CAs \leftarrow CAs$ 
11:  Add  $DM_{SW}.PMVs \leftarrow PMVs$ 
12:  Add  $DM_{SW}.SSRs \leftarrow SSRs$ 
13:  Add  $DM_{SW}.LTLs \leftarrow LTLs$ 
14:  Add  $DataModel_{SW}[i] \leftarrow DM_{SW}$ .
15: end for
16: Return  $DataModel_{SW}$ 

```

have a state decomposition that indicates what type of sub-states the superstate can contain. All sub-states of a superstate $S_{i,j}$ should have the same type of decomposition of the parent state.

4.4.3. Generating the tree of the Stateflow model. The algorithm for generating the tree of the Stateflow (shown in algorithms 2 & 3) starts by parsing the XML specifications of the Simulink's Stateflow Sf into the data model DM_{Sf} (see lines 1 – 3). A tree Stateflow will be created to store a root node, a list of transitions and the list of the Stateflow variables. As a Stateflow model has no root state, a default node called *root* will be created to store all information about the superstates at level 0 and assigned its *ParentID* randomly as an integer number that is not assigned to any state in the Stateflow model (see lines 4 – 7). Each node stores the following data: *id*, *name*, *parentID*, *T* a list of transitions, a list of children (sub-states), the order of execution, a list of the state actions (entry, during and exit actions) and type of decomposition state (*OR State* or *AND State*). All superstates at level 0 in the Stateflow model are added as the children of the default *root* node. For each state $S_{i,j}$, a node will be created to store all information of the state $S_{i,j}$ (see lines 8 – 14). If the state $S_{i,j}$ has children, then all its sub-states will be traversed recursively until no more children exist for the superstate (see line 15). Then, a state *node* will be added as a child of the root node (see line 17). The transitions at this level will be added to a transition list of the Stateflow tree T_{sf} to be used in the next algorithms (see line 21 – 23): the *SMVGenerator* algorithm, Extended Finite State Machine model (*EFSMGenerator*) and a truth-table of the EFSM model generator.

Figure 10 shows the tree of the stateflow model of the train door software controller.

Algorithm 2 Generate a Tree of Stateflow Data**Input:** S_f : A Simulink Stateflow file**Data:** DM_{S_f} = A data model to store all data of Stateflow in S_f , S = A list of states of Stateflow S_f .**Output:** T_{S_f} = a tree which represents all information of Stateflow states $\in S_f$.**Description:**

```

1: URL schemaFile ("/Stateflowschema.xsd")
2: XSMModel LoadXMLSchema (schemaFile)
3:  $DM_{S_f}$  ParseXMLSchema ( $S_f$ )
4: Extract all states at level 0:  $S \leftarrow DM_{S_f}.Stateflow.getStates()$ 
5: Create a state root node  $\leftarrow root$ 
6: Set ParentID  $root \leftarrow ParentID \notin DM_{S_f}.States.IDs$ 
7: Name  $root \leftarrow 'root'$ 
8: for each State  $s$  in  $S$  do
9:   Create a state child node  $\leftarrow node$ 
10:  Set ParentID  $node.parentID \leftarrow root.ID.$ 
11:  Set Data  $node.name \leftarrow s.name$ 
12:   $node.Id \leftarrow s.SSID$ 
13:   $node.setDecomposition \leftarrow s.getDecomposition()$ 
14:   $node.setStatesActions \leftarrow s.getStatesActions()//Entry, During and Exit Actions$ 
15:  if  $s.hasChildren() == true$  then
16:     $node.isHasChildren(true)$ 
17:    traverseChildren ( $node, s$ )
18:  end if
19:  Add  $root.addChild ( node )$ 
20: end for
    // Extract all transitions between the states.
21:  $T_{S_f}.setTransitions(DM_{S_f}.getTransitions())$ 
    // Extract all variables of Stateflow.
22:  $T_{S_f}.setVariables(DM_{S_f}.getVariables())$ 
23:  $T_{S_f}.root = root$ 
24: Return  $T_{S_f}$ 

```

4.4.4. Generating the SMV model from the STPA and Stateflow data models. Figure 11 shows the basic structure of the SMV model as described in [Cavada et al. 2010]. Each SMV module represents a superstate in the Stateflow model which can contain the following sections: 1) The name of the model with the optional state variable parameters, 2) The declaration of the state variable and their possible values, 3) The initial values of variables and the *states* variable, 4) The sub-modules of the super module declaration, 5) The transitions of the module, and a list of the LTL formulae. To represent the states of the Stateflow model (\simeq internal state variables of each controller in STPA) in an SMV model, we declare an enumeration variable called "*states*" which contains the names of sub-states of the superstate in the Stateflow model.

Based on the principles of the SMV model [Cavada et al. 2010] and Stateflow diagram [MathWorks 2016], we develop an algorithm to transform the Stateflow (safe behavioural model) and STPA data objects into an SMV model. Algorithm 4 shows the process of automatically transforming the safe behavioural model and the STPA data models into an input language of the SMV model checker. The algorithm traverses the states of the safe behavioural model recursively and generates the SMV model by parsing the hierarchical levels of the safe behavioural model. The inputs of the algorithm are a tree of Stateflow model T_{S_f} which is created based on algorithm 2 and the STPA

Algorithm 3 `traverseChildren(root, s)`**Input:** *root* : a root node in the tree T_{Sf} , *s*: a state in a satateflow data model DM_{Sf} **Description:**

```

1: if s.hasChildren()==true then
2:   for each State child in s.getChildren() do
3:     Create a new node node
4:     Set node.setName  $\leftarrow$  child.getName
5:     node.setId  $\leftarrow$  child.getID
6:     node.setParentID  $\leftarrow$  child.getParentID
7:     node.setDecomposition  $\leftarrow$  child.getDecomposition()
8:     node.StatesActions  $\leftarrow$  child.StatesActions()//Entry, During, Exit Actions
9:     if child.hasChildren()== true then
10:      node.setHasChildren(true)
11:     end if
12:     Add root.addChild(node)
13:     traverseChildren(node, child)
14:   end for
15: end if

```

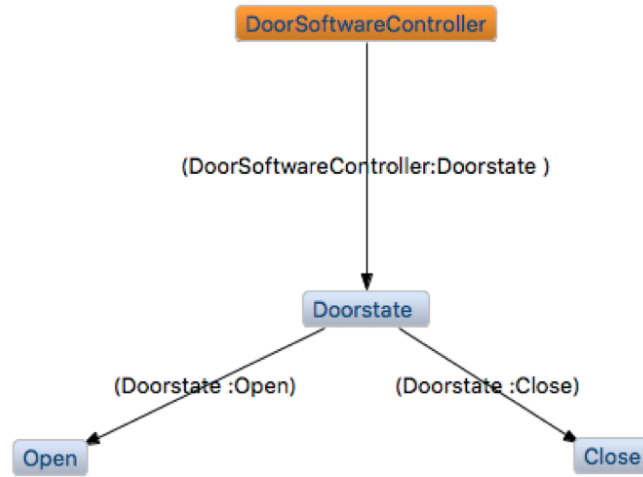


Fig. 10: The tree of the stateflow model of the train door software controller

data model $DataModel_{SW}$ of the software controller CO_i , which is generated based on algorithm 1 and a node n in the tree T_{Sf} .

The algorithm 4–5 process starts by creating an object of the *SMV* model which represents all structure data of the *SMV* model (see line 1). The algorithm takes the root node of the safe behavioural model tree as the input at the first time to create the main module of the *SMV* model, then it create the main module section (see lines 2 – 6) and declares the *VAR* section (see line 7). In this section, the algorithm will declare the local variables of the root node and maps their data types to the *SMV* data types (see lines 8 – 9). The algorithm will check whether the variables are declared exactly in the process model of the software controller in the STPA control structure diagram to reduce the time and effort of matching these variables during the verification step (see line 10). If a name of variable or state in the STPA data model does not match any

```

1  MODULE main (<module variables>)
2    VAR
3      variables : <range data type>/<enumeration>
4      <nameSub1>: _SubModule1 (variables)
5      ...
6      <nameSubN>: _SubModuleN (variables)
7      states: <All children states>
8      ASSIGN
9        INIT (states=<initialState>)
10       INIT (<variable> =<value>)
11       next(<variable>):= case
12       <var1>=<value> & <tranConditon>:<nextValue>;
13       ...
14       next(states):= case
15       <states>=value & transition:nextState;
16       ...
17     esac;
18   LTLSPEC
19     <List of LTL formulae>

```

Fig. 11: The structure of the SMV model is generated from the Stateflow tree and the STPA data object

name in the data model of the safe behavioural model, then the algorithm will show a message to the user and return *null* (see line 49).

The SMV model does not support the same basic data types (int, double, single) as the data types which are declared in the Stateflow model, it supports only a finite range type as integer range *min...max* value. Therefore, the algorithm should map the data types (int, double or single) into a finite range which starts with a minimum value and ends with a maximum value of integer data type. The enumeration data types are declared into the Stateflow model as a class which is saved in a separate file and not in the XML specifications of Stateflow model. Therefore, the algorithm checks each variable with enumeration data type whether it is a process model variable in the STPA data model or not. In case the enumeration variable is a process model variable, the algorithm takes its values as they are defined in the STPA process model variable values. Otherwise, the algorithm creates an empty bracket {} for the values of the enumeration variable and prompts the user to determine the values of this variable (see lines 11 – 15).

Next, the algorithm checks whether the root state *root* has children states and which of them has children too (see line 16 – 20). In case that a child *node* of *root* has children, then the algorithm declares a sub-module for this child *node*. Then, the algorithm takes all variables of the current state *node* to create a list of the parameters of the sub-module (see line 18). Next, the algorithm parses the sub-states of the superstate and creates the variable "states" with a list of the names of the sub-states as values (see lines 21 – 24). The algorithm will create the section "Assign" to initial the states and variables of the SVM model (see line 15). The algorithm will create the *initial* expression of the "states" variable. Each data variable will also be initialised with the minimum value of its data type such as a variable with a numeric data type with zero, Boolean with FALSE and enumeration variable with the first value (see in lines 26 – 27). Next, the algorithm will parse all transitions of the current state *node* and create the *next* expressions for the "states" variable (see lines 28–34). The *next* expressions of *states*

Algorithm 4 generateSMV(T_{sf} , $DataModel_{sw}$, n)

Input: T_{sf} : a tree data model of safe behavioural model, $DataModel_{sw}$: a STPA data model of controller CO_i , n : is a node in tree T_{sf} .

Output: SMV_i : an SMV object represents the data of SMV model.

Description:

```

1:  Create a  $SMV_i \leftarrow$  SMV model object
2:  if ( $n.isRoot() == true$ ) then
3:    Set header of  $SMV_i \leftarrow$  'Module main'
4:  else
5:     $SMV_i \leftarrow$  'Module' root.getName() (root.getVariables())
6:  end if
7:  Set VAR section of  $SMV_i \leftarrow$  'VAR'
8:  Parse Variables  $SMV_i.setVariables() \leftarrow T_{sf}.getVariables()$ 
9:  Map data type of  $SMV$  variables into SMV data types.
10: if (ValidateSTPADataModel( $n.getVariables()$ ,  $DataModel_{sw}$ ) then
11:   if ( $v.getType() \neq "Enum"$ ) then
12:     Declare each variable as  $v.getName : v.getType()$ ;
13:   else
14:      $v.getName : \{\}$ ; // an empty bracket
15:   end if
16:   if ( $n.isSubModule == true$ ) then
17:     for each  $s \in n.getChildren()$  do
18:       Declare "Sub_" +  $s.getName(n.getVariables())$  ( $n.getVariableNames()$ )
19:     end for
20:   end if
21:   Declare states variable in  $SMV \leftarrow$  'states'
22:   for each  $s \in n.getChildren()$  do
23:      $states \leftarrow .s.getName()$ 
24:   end for
25:   Set ASSIGN section of  $SMV \leftarrow$  'ASSIGN'
26:   initial each  $v$  of  $SMV_i \leftarrow$ 
27:    $init(v.getName()) := initial\_Value$ ;
28:   Parse Transitions  $T \leftarrow n.getTransitions()$ 
29:   Set Next section of  $T$  of  $n$  state
30:    $SMV_i \leftarrow$  'next' ( $states$ ) := case
31:   for each  $t \in T$  do
32:      $states := t.Source \& t.Condition : t.Destination$ ;
33:   TRUE:  $states$  ; esac
34:   end for
35:   if ( $n.isRoot() == true$ ) then
36:     for each  $v \in n.getVariables()$  do
37:        $SMV_i \leftarrow$  'next' ( $v.getName()$ ) := case
38:        $states = n.getSource() : n.getEntryDuringExit(v.getFunction())$ 
39:       TRUE:  $v.getName()$  ; esac;
40:     end for
41:   end if

```

Algorithm 5 generateSMV(T_{Sf} , $DataModel_{SW}$, n) (continued)

```

42:    $SMV_i \leftarrow \text{'esac;}'$ 
43:   if ( $n.hasChildren()$ ) then
44:     for  $s \in root.getChildren()$  do
45:        $SMV_i \leftarrow \text{generateSMV}(T_{Sf}, DCs_i, s)$ 
46:     end for
47:   end if
48: else
49:   Show "STPA variables do not match  $Sf$  variables" & Set  $SMV_i \leftarrow \text{null}$ 
50: end if
51:  $SMV_i \leftarrow \text{"LTLSPEC " } DCs_i.getLTL()$ 
52: Return  $SMV_i$ .

```

variable refer to the transition relations of current state *node* with other states in the model (the truth-table). The *next* expressions of the *states* variable are expressed as follows:

```

next(states):= case
states=<sub-state> : <nextstate>
...
1: {All sub-states}; esac;

```

To create the *next* expressions for each data variable, the algorithm parses the *Entry*, *During* and *Exit* actions of the current state and extracts all actions of each variable (see lines 35 – 41). The *next* expressions of the data variables refer to the values of variables in the next state. The *next* expressions of each data variable are expressed as follows:

```

next(variable):= case
states = <state> & transition: <nextValue>
....

```

The algorithm will continue parsing the superstate in the tree of the safe behavioural model (Stateflow) till all superstates have been visited (see lines 43–48). The generated SMV specifications of each sub-module and the main module will be saved as a string into a stack object. Finally, the algorithm will fetch the LTL formulae from the STPA data model object and add them at the end of the main-module section (see lines 50–51).

To check the correctness of the generated SMV model and the safe behavioural model, we run the NuSMV model checker to verify whether the SMV model contains errors and verify it against the STPA software safety requirements expressed in the LTL formulae and saved to the SMV model.

Figure 12 shows an example of the generated SMV model of the safe behavioural model of the train door software controller.

4.5. Automatically Constructing the Safe Test Model from the Safe Software Behavioural Model

After ensuring the correctness of the generated SMV model of the safe behavioural model (Stateflow model), the safe behavioural model which uses the notations of the Simulink's Stateflow should be transformed into the EFSM notation. For this purpose, we develop an algorithm to map the Stateflow tree of the safe behavioural model and

```

1  MODULE Sub_Doorstate (Trainposition,Trainstatus,
    PersonIndoorway)
2  VAR
3    controlAction:{Opendoor,Closeddoor,Stop};
4    states: {Open,Close};
5  ASSIGN
6    init (states):=Open;
7  next (states):=case
8    states=Open & (Trainposition=Aligned & Trainstatus=Stop):
        Open;
9    states=Close & (PersonIndoorway) : Open;
10   states=Open & (!PersonIndoorway) : Close;
11   states=Close & (Trainposition=Aligned & Trainstatus=Stop) :
        Close;
12  TRUE: {Open ,Close };
13  esac;
14
15  MODULE main
16  VAR
17    PersonIndoorway: boolean;
18    Trainposition: {Aligned, NotAligned}
19    Trainstatus: {Stop, Move}
20    Doorstate :Sub_Doorstate (Trainposition,Trainstatus,
        PersonIndoorway);
21  states: {Doorstate};
22  ASSIGN
23    init (states):=Doorstate ;
24    init (PersonIndoorway) := FALSE ;
25  next (states):=case
26  TRUE:{Doorstate};
27  esac;
28  LTLSPEC G (((trainstatus== stop) & (doorstate == close) & (
        trainposition== Aligned) & (PersonIndoorway==TRUE)) ->
        !(controlAction== closedoor))

```

Fig. 12: An example of the generated SMV model for the train door software controller

its truth-table into an EFSM model. The algorithm 6–7 shows the process of transforming the tree of the Stateflow model into an EFSM model. The idea here is to eliminate the hierarchical and concurrent structure of the Stateflow model (flattened and broadcast communication) and transform them into the EFSM notations by considering the state decomposition (exclusive or parallel).

The algorithm 6–7 starts by taking the root node of the Stateflow tree T_{sf} as the root node of the EFSM model and the truth-table of the Stateflow as the truth-table of the EFSM model (see line 1). The Stateflow semantic supports multi-hierarchy levels of states, whereas the EFSM model does not. Therefore, the truth-table of the EFSM model must not have any source or destination node as a superstate (a state that has children). The idea here is to investigate the truth-table of Stateflow and update the destination and source parent state with its sub-states. At the beginning, the algorithm checks whether there is a superstate in the truth-table (see lines 2–3). For each transition $t \in T$ in the truth-table, the algorithm will identify its source and destina-

Algorithm 6 GenerateEFSM (T_{sf})**Input:** T_{sf} : a tree of Stateflow model,**Output:** $EFSM$: a Java object represent all data of EFSM**Description:**

```

1: Create StateNode  $root \leftarrow T_{sf}.getRoot()$ 
2: Get TruthTable  $truthTable \leftarrow T_{sf}.getTruthTable()$ 
3: if  $root.hasChildren() == \text{ture}$  then
4:   Set Initial state  $\leftarrow T_{sf}.getInitialState()$ 
5:   while  $isHasSuperState(truthTable)$  do
6:     for Transition  $t \in truthTable$  do
7:       StateNode  $src \leftarrow t.getSourceNode()$ 
8:       StateNode  $dest \leftarrow t.getDestinationNode()$ 
9:       if  $src.isSuper() \& \neg(dest.isSuper())$  then
10:         $get\ children \leftarrow src.getChildren()$ 
11:        for  $child \in children$  do
12:           $updateTruthTable(child, dest, t, truthTable)$ 
13:        end for
14:      else
15:        if  $\neg(src.isSuper()) \& dest.isSuper()$  then
16:          if  $dest.Decomp('AND\_STATE')$  then
17:             $get\ children \leftarrow dest.getSubSates();$ 
18:            for  $child \in children$  do
19:               $updateTruthTable(src, child, t, truthTable)$ 
20:            end for
21:          else
22:            if  $dest.Decomp('OR\_STATE')$  then
23:               $S_D \leftarrow getDefaultState(dest)$ 
24:               $updateTruthTable(src, S_D, t, truthTable)$ 
25:            end if
26:          end if
27:        end if
28:        if  $src.isSuper() \& dest.isSuper() \& dest.Decomp('OR\_STATE')$ 
then
29:           $get\ srcChildren \leftarrow src.getSubSates();$ 
30:           $get\ def \leftarrow dest.getDefaultState();$ 
31:          for  $s \in srcchildren$  do
32:             $updateTruthTable(s, def, t, truthTable)$ 
33:          end for
34:        end if

```

tion states and create two state nodes (see lines 6–8). Next, the algorithm will check their state decomposition as follows:

- If source state $src \in T_{sf}$ of transition t is a **superstate** with a state decomposition “OR.STATE” or “AND.STATE” and the destination node $dest \in T_{sf}$ is **not superstate**. Each sub-state of src state must be linked to the destination state $dest$ by creating a new transition with the same information of transition $T \in T_{sf}.TruthTable$ for each sub-state and only update the source with sub-state (see lines 9 – 14).
- If source state $src \in T_{sf}$ is **not superstate** and the destination state $dest \in T_{sf}$ is **superstate** with a state decomposition “AND.STATE”. All sub-states of $dest$ state should be identified and linked with the source state (see line 15 – 21). Algorithm

Algorithm 7 GenerateEFSM (T_{sf}) (continued)

```

35:         if  $src.isSuper()$  &  $dest.isSuper()$  &  $dest.Decomp('AND\_STATE')$ 
      then
36:             get  $srcChildren \leftarrow src.getSubSates();$ 
37:             get  $destChildren \leftarrow dest.getSubSates();$ 
38:             for  $s \in srcchildren$  do
39:                 for  $d \in destchildren$  do
40:                      $updateTruthTable(s, d, t, truthTable)$ 
41:                 end for
42:             end for
43:         else
44:             if  $!(src.isSuper()) \& !(dest.isSuper())$  then
45:                  $updateTruthTable(src, dest, t, truthTable)$ 
46:             end if
47:         end if
48:     end if
49: end for
50: end while
51: end if
52: Add  $EFSM.setTruthTable \leftarrow truthTable$ 
53: Add  $EFSM.setStates \leftarrow T_{sf}.getStates()$ 
54: Return  $EFSM$ .

```

7 will create a new transition for each sub-state of $dest$, where source is src and destination is the sub-state of destination.

- If source state $src \in T_{sf}$ is **not superstate** and the destination state $dest \in T_{sf}$ is **superstate** with a state decomposition “OR_STATE”. The default state $defaultState$ of superstate $dest$ (a default state is a state which has a default transition) should be identified (see lines 22 – 27). Algorithm 7 will create a new transition and set its source as src and its destination as the default state of destination.
- If source state $src \in T_{sf}$ is **superstate** with a state decomposition “OR_STATE” or “AND_STATE” and the destination state $dest \in T_{sf}$ is **superstate** with a state decomposition “OR_STATE”. All sub-states of src state should be identified and linked with a default state of $dest$ state (see lines 28 – 34). Algorithm 7 will create a new transition for each sub-state of src and its source is src and its destination is the default state of destination $dest$ state.
- If source state $src \in T_{sf}$ is **superstate** with a state decomposition “OR_STATE” or “AND_STATE” and the destination state $dest \in T_{sf}$ is **superstate** with a state decomposition “AND_STATE”. All sub-states of src state should be identified and linked with all sub-states of $dest$ state (see lines 35 – 43). Algorithm 7 will create a new transition for each sub-state of src and its source is src and its destination is the sub-state of destination $dest$ state.
- If source state $src \in T_{sf}$ is **not superstate** and the destination state $dest \in T_{sf}$ is **not superstate**. A transition t will be added into the truth-table (see lines 44 – 46).

The algorithm runs continuously till no superstate exist in the truth-table. All sub-states (without children) in the Stateflow model tree will be taken as the states of the EFSM model. Also, all data variables of the Stateflow model and the actions of the state (entry, exist, during) will be added into the states of EFSM.

Figure 13 shows an example of the generated EFSM of the train door software controller

Algorithm 8 UpdateTruthTable (*src*, *dest*, *t*, *truthTable*)

Input: *src* : a source node of transition *t*, *dest*: a destination node of transition *t*, *t* : a transition in the truth-table, *truthTable*: a truthTable of Stateflow tree T_{sf}

Description:

- 1: **create** new Transition *t_{new}*
- 2: **set** data *t_{new}* $\leftarrow t$
- 3: **update** *t_{new}*.setSrc(*src*)
- 4: **update** *t_{new}*.setDest(*dest*)
- 5: **add** *truthTable* $\leftarrow t_{new}$

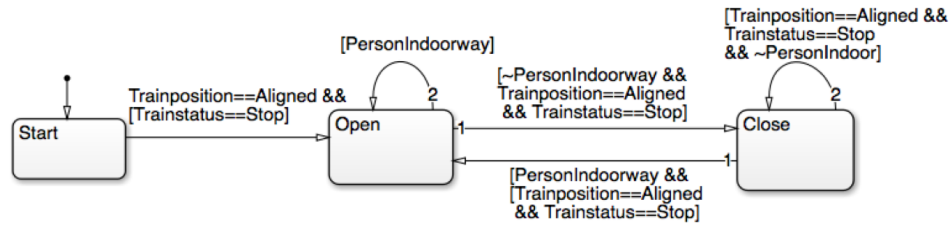


Fig. 13: The safe test model (EFSM) of the train door software controller

4.6. Automatically Generating Safety-Based Test Cases

The final step is to generate the test cases from the safe test model (extended finite state machine) which are constructed from the safe behavioural model.

We developed a random walk-based algorithm for automatic test case generation from the safe test model. We implemented three search-based algorithms (e.g. depth-first search, breadth-first search, and both combined depth-breadth-first search). The idea behind here is to select a state in the safe test model as a start node and transform into a Java Script function at run time. The Java script function takes the variables which are declared in the state actions (Entry, during, Exit) of each as parameters and executes the state actions to update the values of the variables. The return value of the function will be determined based on the data type of each variable which is declared in the Simulink Stateflow model. Next, the algorithm will check the transition conditions of a state to determine which is the next state. During traversing the safe test model, the information of the visited states (path sequences) will be saved in a test suite.

Generating test cases from a model usually leads to an infinite number of possible test cases. Therefore, it is necessary to choose a suitable test coverage criteria to manage the generating process. In our algorithm, we identify three test coverage criteria: 1) *state coverage* which is the number of visited states divided by the total number of the states of the model, 2) *transition coverage* is the number of the executed transitions divided by the total number of the transitions, 3) *STPA safety requirements coverage* in which each STPA software safety requirement should be covered at least in one test case to trace how the STPA-generated software safety requirements are covered into the generated test cases. To measure the STPA SSR coverage, we define a *safety requirements traceability* matrix between the generated safe test model and STPA software safety requirements to manage the quality of the test case generating process and measure the coverage of STPA software safety requirements in the generated safety-based test cases. As the safe test model of the safe behavioural model is constrained with STPA safety requirements (step 2) and contains the process model variables as states, the algorithm will automatically generate the traceability matrix

Algorithm 9 Generate Traceability Matrix ($SSR, TN, src, minSimilarity$)

Input: SSR : a STPA-generated software safety requirement, TN : a transition condition in a safe test model extracted from SBM . src : a source node of transition condition Tn .

$minSimilarity$: a minimum degree of similarity between 5% ... 100%.

Output TM : a traceability matrix. **Description:**

```

1: Add  $TN \leftarrow states = src.getName()$ 
2: Add  $TN \leftarrow controlAction = src.getAction().getName()$ 
3: tokenize  $SSR[] \leftarrow SSR$ 
4: tokenize  $TN[] \leftarrow TN$ 
5: get  $max\_Tokens \leftarrow \max(SSR[], TN[])$ 
6: inital  $Sim \leftarrow 0$ 
7: inital  $matched\_Tokens \leftarrow 0$ 
8: inital  $i \leftarrow 0$ 
9: while  $i < max\_Tokens$  do
10:   inital  $j \leftarrow 0$ 
11:   while  $j < max\_Tokens - 1$  do
12:     if  $SSR[i] == TN[j]$  then
13:        $matched\_Tokens = matched\_Tokens + 1$ 
14:     end if
15:      $j = j + 1$ 
16:   end while
17:    $i = i + 1$ 
18: end while
19:  $Sim_{SSR, TN} = (matched\_Tokens / max\_Tokens) \times 100$ 
20: if  $Sim_{SSR, TN} \geq minSimilarity\%$  then
21:   Add  $TM \leftarrow SSR \times TN$ 
22: end if
23: Return  $TM$ .
```

($TM = SSR \times TN$, where $SSR \in DCs$ of the STPA data model and TN transition conditions $\in T_{sf}$).

Algorithm 9 shows how to generate the traceability matrix TM by calculating the similarity degree between each STPA-generated software safety requirement(SSR) and the transitions condition (TN) of the safe behavioural model and the input state actions of the source state of transition condition TN . The similarity degree is calculated by the following equation:

$$Sim_{(SSR, TN)} = \frac{|\#Total\ No.\ matched\ tokens\ between\ (SSR, TN)|}{|\#Max\ No.\ tokens\ in\ (SSR, TN)|} \times 100 \quad (1)$$

Algorithm 9 takes a STPA-generated software safety requirement (SSR), a transition condition TN , the source state of the transition condition TN and a minimum degree of similarity ($maxSimilarity$) which should be between 5% ... 100% and entered by the user. To compare between the STPA-generated software safety requirements and transition conditions, the algorithm construct at first the full transition information by including the name of source state and the control action which is provided in this state to the transition condition (shown in Fig. 14). The algorithm creates the full transition information by adding the source state src and the control action $controlAction$ to the transition condition TN . The full transition condition will be constructed as follows:

STPA Software Safety Constraint

Control Action **close door** must be provided when door **state = Open**,
Trainposition = Aligned, Trainstatus = Stop and PersonInDoorway = False

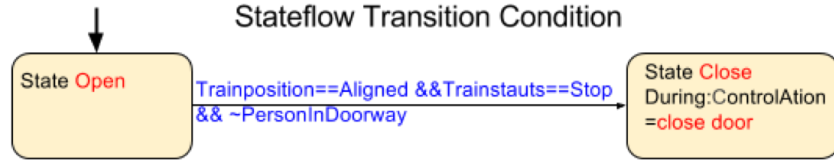


Fig. 14: An example of the similarity degree between STPA safety requirements and Stateflow transition conditions

$fulltransition \leftarrow \{states==src.getName() \text{ and } controlAction==src.getAction() \text{ and } src.getTransitionCondition(TN)\}$

The algorithm calculates the similarity degree based on the equation 1. If the similarity degree is greater than the minimum degree of similarity, then the algorithm will create an item in the traceability matrix for the software safety requirement *SSR* and the transition condition *TN*. The algorithm also allows the user to set the maximum similarity degree between 5..100% before generating the safety-based test cases.

Algorithm 10 shows how to generate the test cases from the safe test model. It takes the generated Safe Test Model (*STM*), a Traceability Matrix *TM*, a list of the test coverage criteria *CC*, a number of test steps which is the total number of executions of the algorithm and a stop condition which is a test coverage criteria to stop the execution of the algorithm when it reaches 100%. The process of generating the test cases from the safe test model can be described as follows:

- (1) The algorithm starts by selecting a random state as the start state and a state as the end state from the safe test model to generate all possible paths between them (see lines 3 – 4).
- (2) A new test suite *ts* will be created to store all the generated test cases.
- (3) Generate for each input data variable a random value between its minimum and maximum values which are identified by the user (see line 7).
- (4) Walk randomly by using the depth-first algorithm, all possible paths between the start and end states will be identified. The path here means a sequence of the visited states and their transitions. We also use the breadth-first algorithm combined with depth-first algorithm to identify all possible paths *PT* from start state to achieve a good test coverage criteria (see line 8 – 11).
 - For each transition *t* in path $pt \in PT$, its transition condition will be transformed into a Java Script function. The test input variables *in* will be passed as an input of a Java Script function. To execute this function at the run time, we use the Java Script Engine which invokes the function with values of input data parameters and returns the result.
 - For each state *s* in path *pt*, the state actions (Entry, During, Exit) will be eliminated and transformed into Java Script functions. These will be executed to update the values of each local *loc* or output variable *out* of each state.
 - Create a new test case *tc*. Each test case will store the information about the sequence path *pt* such as: *id* is a number of the test case, *id_Ts* which is the number of the test suite, *id_SSR* which is the number of the software safety requirement, *preconditions and actions* which is the sequence of the local variables of states in

Algorithm 10 Generate Safety-based Test Cases(STM , TM , CC , $TestSteps$, $StopConiditon$)

Input: STM : a safe test model extracted from SBM , TM : a traceability matrix, CC : is a list of the test coverage criteria, $TestSteps$ is the total number of execution algorithms, $StopCondition$: a condition to stop the execution process.

Output TS : a list of test suites, each test suite should contain a list one test case TC .

Description:

```

1: Initial step  $\leftarrow 0$ 
2: while step  $< TestSteps$  do
3:   Choose start state  $\leftarrow STM.getRandomState()$ 
4:   Choose end state  $\leftarrow STM.getRandomState()$ 
5:   Create a new test suite  $ts$ 
6:   if  $StopConiditon < 100.0\%$  then
7:     Randomly Generate_Tes.InputData ()
8:     Walk  $TC_i \leftarrow GenerateTestCasesByDFS$  (start, end)
9:     Add  $ts \leftarrow TC_i$ 
10:    Walk  $TC_j \leftarrow GenerateTestCasesByBFS$  (start)
11:    Add  $ts \leftarrow TC_j$ 
12:   else
13:     if  $StopConiditon == 100.0\%$  then
14:       Calculate_Coverage_Criteria()
15:       STOP
16:     end if
17:   end if
18:   ADD  $TS \leftarrow ts$ 
19:   Calculate_Coverage_Criteria()
20:   unvisitedTransitions( $STM$ )
21:   unvisitedStates( $STM$ )
22:   Initial step  $\leftarrow$  step + 1
23: end while
24: Return  $TS$ .
```

the path pt and their updated values, and $postconditions$ which is the sequence of output variables and their values.

- (5) Check whether the test case tc has been covered in any test suite. If it hasn't, tc will be added to the test suite ts (see line 13).
- (6) Calculate the test coverage criteria and check the stop condition of the algorithm (see line 14).
- (7) Change status of all states and transitions in the safe test model to unvisited to generate a new sequence path (see line 20 – 21).
- (8) The algorithm will be continued (repeat 1-8) till the stop condition is achieved (100%) or the number of executions the algorithm has been reached to the total number of the test steps.

Ultimately, the time spent during test case generation process, the values of the test coverage criteria and a list of test suits and their test cases with the related software safety requirements will be automatically saved into a CSV file.

5. TOOL SUPPORT

Here we describe the implementation of the proposed approach for generating test cases based on the information derived from the STPA safety analysis. We use the

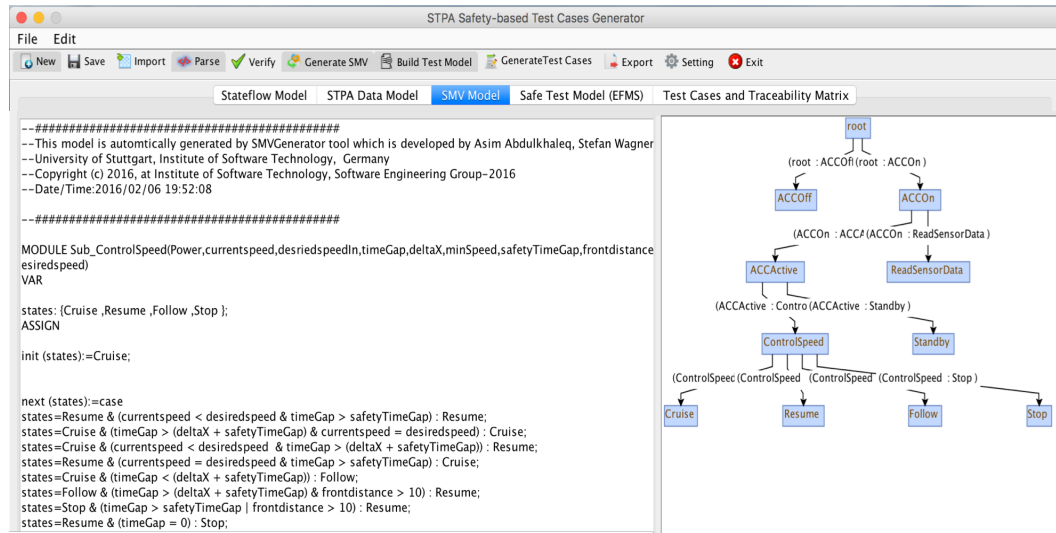


Fig. 15: STPA TCGenerator: STPA test cases generator tool

previous algorithms and rules as the basis for implementing tool support for our safety-based test case generation approach.

To automatically formalize the STPA software safety requirements which are documented in XSTAMPP and transformed into LTL based on rules 1–4, we developed an Eclipse plug-in called XSTPA⁶ based on the XSTAMPP architecture. XSTPA automatically generates the context tables (combinations between process model variables) by using a Java library for the combinatorial testing algorithm called ACTS⁷ [Kuhn et al. 2013] which was developed by the American National Institute of Standards and Technology to generate combination sets of t parameters with n values. Based on rules 3 and 4, XSTPA automatically generates the LTL formulae. The generated LTL formulae will be used to check the correctness of the constructed safe test model which will be used to generate the safety-based test cases for each STPA-generated software safety requirement.

To generate the test cases based on STPA results, we implemented a tool support called *STPA Test Cases Generator* (STPA TCGenerator, shown in Fig. 15) which parses the STPA file project created in XSTAMPP and the safe behavioural model which is created with Simulink's Stateflow editor to generate the SMV model and check the correctness of the safe behavioural model, eliminate the safe test model and generate safety-based test cases. In the following, we summarise the main functions of the STPA TCGenerator:

- Parse the STPA data model which is documented in XML specification into Java objects.
- Parse the XML specification of the Simulink Stateflow model into Java objects.
- Based on the STPA data model and the Simulink Stateflow model, the tool automatically generates the SVM model.

⁶<http://www.iste.uni-stuttgart.de/se/werkzeuge/xstpa.html>

⁷<http://csrc.nist.gov/groups/SNS/acts/index.html>

- Check the consistency between the STPA data model and the specification of Simulink Stateflow and provides the results to the user (e.g. matched, does not match, and unknown).
- Verify the generated SMV model against the generated LTL of the STPA safety requirements.
- Transform the Simulink Stateflow model into the extended finite state model for testing purposes.
- Generate the tractability matrix between STPA safety requirements and the Simulink Stateflow specifications.
- Allow the user to enter the test input data for each input variable.
- Allow the user to configure the test case generation process by adding a number of test steps and selecting the test case generation algorithm and the test coverages.

To support software and safety engineers who use the XSTAMPP platform, we developed an Eclipse plug-in for the STPA TCGenerator tool called *STPA TCGenerator-Plugin* which is integrated into the XSTAMPP platform to generate the test cases for each STPA software safety requirement within the XSTAMPP platform.

The first prototype of the *STPA TCGenerator* standalone version and the results of the illustrative example are available online in our repository⁸. The updatesite of the STPA TCGeneratorPlugin are available online in our repository⁹.

6. AN ILLUSTRATIVE EXAMPLE: A SIMULATOR OF THE ACC SYSTEM WITH STOP AND GO FUNCTION

To illustrate the proposed approach, we developed a simulator software written in ANSI-C to simulate the adaptive cruise control system with stop and go function by using two LEGO EV3 Mindstorm robots¹⁰. The simulator was developed by a bachelor student within 6 months. The ACC with stop and go function [Venhovens and Naab 2000] is an extended version of the normal adaptive cruise control system. It maintains a certain speed and keeps a safe distance from the vehicle ahead based on the radar sensors. The ACC with stop and go function will bring the vehicle to a complete stop when the vehicle ahead comes to a standstill or there is a stationary object in the lane.

Figure 16 shows the mechanism of the simulator of the ACC with stop-and-go function. The ACC simulator maintains a constant time gap to vehicles ahead. It uses a forward ultrasonic sensor with a range of up to 255 centimeters, which is located in the front of the robot to detect the distance of the robot ahead of it and can automatically maintain the pre-set time gap. It adjusts the robot speed by increasing or decreasing the value of current speed to keep a safe distance. If the robot ahead is completely stopped, then the ACC simulator will slow down the robot vehicle to a standstill. If the vehicle ahead starts moving again, then the ACC simulator will automatically start to move again and maintain a constant time gap between the robot ahead. Our simulator algorithm is the ACC simulator starts first read the distance data from the ultrasonic sensor and then computes the time gap by using the following equation:

$$currentTimegap = \left\lceil \frac{Frontdistance}{CurrentSpeed} \right\rceil \quad (2)$$

Second, the simulator computes the standstill time, which is the time at which the ACC vehicle must decrease the speed or stop when the vehicle ahead is close or fully

⁸<https://sourceforge.net/projects/stpastgenerator/>.

⁹<https://sourceforge.net/projects/stpatcgeneratorplugin/>

¹⁰<http://www.iste.uni-stuttgart.de/se/forschung/werkzeuge/acc-simulator/>

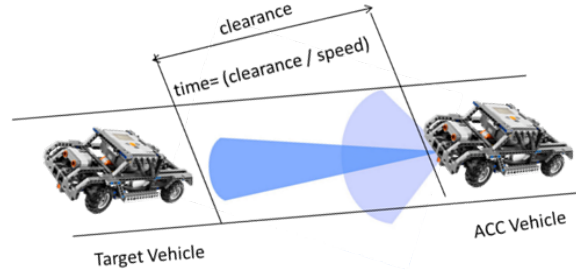


Fig. 16: A mechanism of the simulator of ACC with Stop and Go function

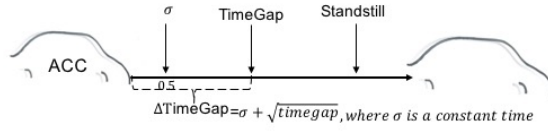


Fig. 17: The ACC system with a Stop and Go function scenarios

stopped. It is calculated as

$$\Delta Timegap = stillstandtime + \sqrt{currentTimegap} \quad (3)$$

Third, the simulator will compare the value of the time gap with the following scenarios (shown in Fig. 17):

- $TimeGap > (\Delta TimeGap + safeTimeGap)$. This indicates that the vehicle ahead is so far from the point t_σ . The simulator will accelerate the speed of the vehicle robot till the desired speed. The simulator adjusts (increase/decrease) the current speed by using the following equation:

$$currentSpeed + / - = \sqrt{speed^2 + 2 * (Time)}, \quad (4)$$

where $Time = ((\Delta Timegap + safeTimeGap) - TimeGap)$

- $(TimeGap > safeTimeGap) \&\& (timeGap < (\Delta TimeGap + safeTimeGap))$. This indicates that the vehicle robot ahead is approaching within the period of time gap between $[t_\sigma \ t_{safeTimeGap}]$. The simulator will put the ACC system in *follow* mode. *Follow* mode means that there is a vehicle in front in the lane. The simulator will automatically adjust the current speed by using equation 3.
- $TimeGap == safeTimeGap$. This indicates that the vehicle robot ahead is approaching within the desired time gap and there is a safety distance between them. The simulator will put the ACC system in the *cruise* mode. *Cruise* mode means that the vehicle robot ahead is approaching in safe time gap. Then, the simulator will set the current speed as the desired speed.
- $TimeGap < safeTimeGap$. This indicates that the vehicle ahead is moving within the time between $[t_{safeTimeGap} \ t_0]$. The simulator will reduce the speed of the vehicle by using equation 3.
- $TimeGap == 0$. This indicates that the vehicle ahead has fully stopped. Then the simulator will bring the vehicle to a complete stop at the standstill distance and change the ACC mode to stop. If the front vehicle starts to move again, then the simulator will change the ACC mode to resume. *Resume* mode means that the current speed of the ACC vehicle will be accelerated to the desired speed. The simulator

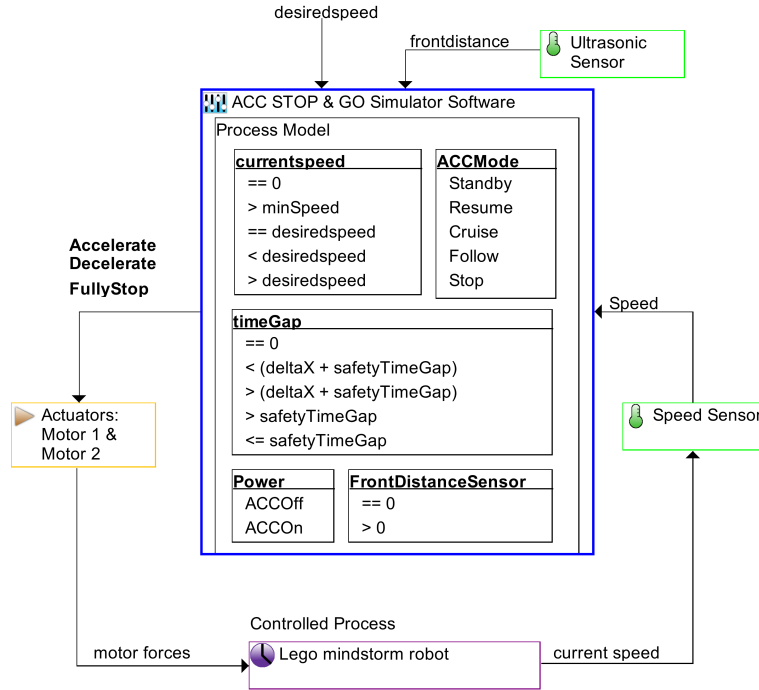


Fig. 18: The control structure diagram of ACC with the safety-critical process model variables

uses the following equation to achieve that:

$$currentSpeed+ = accelerationratio, \quad (5)$$

where accelerationratio is set to 4 cm/sec;

6.1. Deriving Software Safety Requirements of the ACC Simulator

To derive the software safety requirements, we applied the STPA SwISs Step 1 to the system specification requirements. We used the XSTAMPP software tool to document the results of STPA and generate the formal specification of the STPA results.

As a result, we identified the system-level accidents that the simulator software can lead (or contribute to). For example, **ACC-1 : The ACC robot crashes the robot ahead**. The system-level hazards which can lead to this accident are:

- H_1 : The ACC software does not keep a safe distance from the a vehicle robot ahead.
- H_2 : The ACC software provides an unintended acceleration when the vehicle in front is too close.
- H_3 : The ACC software does not stop the vehicle when the vehicle ahead is fully stopped.

We built the control structure diagram of the ACC simulator (shown in Fig. 18). It contains the main interconnecting components of the ACC simulator at a high level, such as the *ACC simulator software controller unit*, the *electronic motors*, the *robot vehicle* as the controlled process, and the *Ultrasonic and speed sensors*. The ACC software controller receives the distance data from the ultrasonic sensor and current speed data from the speed sensor. Based on this information, the software will calculate the time

Table II: Examples of potentially unsafe control actions of the ACC software controller

| Control Action | Not providing causes hazard | Providing causes hazard | Wrong timing or order causes hazard | Stopped too soon or Applied too long |
|------------------|--|---|---|---|
| Accelerate Speed | The ACC software does not accelerate the speed when the robot vehicle ahead is so far in the lane. [Not Hazardous] | UCA-1.1: The ACC software accelerates the speed of robot unintendedly when the time gap to the robot vehicle ahead is smaller than the desired time gap. [H-1] [H-2] | UCA-1.2: The ACC software accelerates the speed before the robot vehicle ahead starts to move again. [H-1] [H-2] | UCA-1.3: The ACC software accelerates the speed too long so that it exceeds the desired speed of the robot. [H-2] |
| Decelerate Speed | UCA-1.5: The ACC software does not decelerate the speed when the robot vehicle ahead is too close in the lane. [H-1] | The ACC software decelerates the speed of robot unintendedly when the time gap to the robot vehicle ahead is larger than desired time gap. [Not Hazardous] | The ACC software decelerates the speed when the robot vehicle ahead starts to move again. [Not Hazardous] | UCA-1.6: The ACC software decelerates the speed long enough so that it cannot bring the robot to fully stop when the robot ahead is stopped. [H-3] |
| FullyStop | UCA-1.4: The ACC software does not bring the robot to a complete stop at a standstill when the robot vehicle ahead is fully stopped. [H-1, H-3] | The ACC software stops the robot suddenly when the distance to the robot ahead is too far. [Not Hazardous] | The ACC software does not accelerate the speed after the robot vehicle ahead starts to move again. [Not Hazardous] | N/A |

Table III: Examples of software safety requirements at the system level

| Related UCAs | Corresponding Safety Constraints |
|--------------|---|
| UCA-1.1 | SSR1.1- ACC software must not accelerate the speed of the robot when the target robot vehicle is too close in the lane. |
| UCA-1.2 | SSR1.2- ACC software must not accelerate the speed when the robot ahead is fully stopped. |
| UCA-1.3 | SSR1.3- ACC software must not increase the speed than the desired speed. |
| UCA-1.4 | SSR1.4- ACC controller must stop the robot at standstill point (shown in Fig. 17) when the robot ahead is fully stopped. |

gap and determine if the vehicle robot ahead is present. The ACC software will adjust the speed of the robot based on the above sensors and issues one of the critical safety control action: accelerate, decelerate, or fullystop. Each one of these control actions will be evaluated based on the four general hazardous types (columns of table II). Table II shows the examples of the potential unsafe control actions of the ACC simulator.

We evaluated each item in table II to check whether it can contribute or lead to any system-level hazards ($H_1 - H_3$). If an item is hazardous, then we assign one or more system-level hazards to it. We translate each hazardous item manually to the corresponding software safety requirement by using the guide words, e.g., *have to*, *must be*, or *should*. Table III shows examples of the informal textual software safety requirements.

Table IV: Examples of the context table of *providing* the control action *accelerate*

| Control Actions | Process Model Variables | | | Is it a hazardous Control Action? |
|-------------------|-------------------------|--|----------|-----------------------------------|
| accelerate | CurrentSpeed | TimeGap | ACC Mode | providing |
| | CS > minSpeed | TimeGap < (Δ Timegap + safetyTimeGap) | follow | No |
| | CS ≤ desiredSpeed | TimeGap == 0 | follow | Yes, H2, H1 |
| | CS < desiredSpeed | TimeGap > safetyTimeGap | follow | No |
| | CS < desiredSpeed | TimeGap < (Δ TimeGap + safetyTimeGap) | follow | Yes |

Table V: Examples of unsafe software scenarios in XSTAMPP based on critical combinations

| ID | Unsafe software safety scenarios |
|----------|---|
| RUCA-1.1 | The ACC software controller provides the accelerate command when ACC mode is Standby and timeGap is greater than (deltaX + safetyTimeGap) and the current speed is less than desired speed. |
| RUCA-1.2 | The ACC software controller provides the accelerate command when timeGap is less than (deltaX + TimeGap). |
| RUCA-1.3 | The ACC software controller provides the accelerate command when current speed is greater than or equal to desired speed. |
| RUCA-1.4 | The ACC software controller does not provide the fullyStop command when the timeGap is 0. |
| RUCA2.1 | The ACC software controller provides the decelerate command too late when ACC mode is follow and timeGap is less than safetyTimeGap and currentSpeed is greater than desired speed. |

To refine the informal textual software safety requirements which are shown in table III, we identified the process model of the ACC software controller and its critical variables which have an effect on the safety of the ACC software control actions. Figure 18 shows the control structure diagram and process model variables of the ACC software. The ACC software has three safety-critical process model variables: *Internal variables* such as currentSpeed (5 values), Timegap (5 values), *Internal states variable* such as ACC mode (states) with 5 values, and *the environmental variables* such as front distance. Each safety control action provided by the ACC software should be evaluated to determine whether it will be hazardous or not when the combination set of relevant values of the process model variables (context) occur.

We used XSTAMPP to generate the critical combinations (context tables) for each safety-critical action in the two contexts *when the control action is provided* and *it is not provided* and causes hazard. For each control action, the total number of combinations between the process model variables is $(5 \times 5 \times 5 = 125)$ combinations. We reduced the number of combinations by applying pairwise test coverage to the generated combination sets. The number of critical combinations is reduced to 25 for each control action.

Based on the generated combination sets, we evaluated each control action in two contexts *Providing* and *Not Providing*. Table IV shows examples of the context table of providing the control action *accelerate* based on the combinations of the values of the critical process model variables. As a result, we identified 32 unsafe scenarios (shown in Table V) for all the control actions *accelerate* (18 scenarios), *decelerate* (7 scenarios)

Table VI: Examples of generated software safety requirements in XSTAMPP for the unsafe scenarios

| Related UCAs | Refined Safety Constraints |
|--------------|--|
| RUCA-1.1 | RSSR1.1 - Accelerate command must not be provided when ACC mode is Standby and timeGap is greater than (deltaX + safetyTimeGap) and the current speed is less than desired speed. |
| RUCA-1.2 | RSSR1.2 - Accelerate command must not be provided when timeGap is less than (deltaX +TimeGap). |
| RUCA-1.3 | RSSR1.3 -Accelerate command must not be provided when current speed is greater than or equal to desired speed. |
| RUCA-1.4 | RSSR1.4 -FullyStop command must provided when the timeGap is 0. |
| RUCA-2.1 | RSSR2.1 - Decelerate command must not be provided too late when ACC mode is follow and timeGap is less than safetyTimeGap and currentSpeed is greater than desired speed. |

Table VII: Examples of LTL formulae of the refined software safety requirements at the system level

| Refined SSRs | Corresponding LTL formula |
|--------------|---|
| RSSR1.1 | LTL1.1 - $G ((state=Standby) \ \&\& \ (timeGap > \delta X + safetyTimeGap) \ \&\& \ (currentSpeed < desiredSpeed) \rightarrow \neg (controlAction == Accelerate))$. |
| RSSR1.2 | LTL1.2 - $G((currentSpeed > desiredSpeed) \ \&\& \ (TimeGap < (\delta Time + safetyTimeGap)) \rightarrow \neg (controlAction == Accelerate))$. |
| RSSR1.3 | LTL1.3 - $G((currentSpeed \geq desiredSpeed) \rightarrow \neg (ControlAction == stop))$. |
| RSSR1.4 | LTL1.4 - $G((timeGap == 0) \rightarrow X (controlAction == FullyStop))$. |
| RSSR2.1 | LTL2.1 - $G ((state == Follow) \ \&\& \ (timeGap < safetyTimeGap) \ \&\& \ (currentSpeed \geq desiredSpeed) \rightarrow \neg (controlAction == Decelerate))$ |

and *FullyStop* (7 scenarios). Table VI shows the examples of generated software safety requirements for the unsafe scenarios.

6.1.1. *Formalizing the Software Safety Requirements of the ACC Software Simulator.* We formalised the STPA-generated software safety requirements of the ACC software simulator which are derived in Step 1 of the proposed approach (shown in Table VI). We used XSTAMPP to automatically refine the informal textual software safety requirements into formal textual software safety requirements (shown in Table VI). Based on the rules 3-4, XSTAMPP automatically generates the LTL formula for each refined software safety requirement. Table VII shows the examples of the corresponding LTL formula of each software safety requirement. We used the generated-LTL formulae to verify the safe behavioural model which is constructed from the STPA results.

6.2. Automatically Generating SMV Model

We visualised the process model of the ACC software controller (shown in Fig. 18) by creating a Simulink/Matlab Stateflow model (shown in Fig. 19). The Stateflow contains 9 states (2 of them are superstates) and 19 transitions. It shows the relationship between the process model variables in the safety control structure diagram of the ACC simulator. The process model describes the critical variables and states of the software and how the software issues the critical safety control actions (e.g. accelerate, decelerate, etc.)

We generated the SMV model of the safe behavioural model (shown in Fig. 18) by using the STPA TCGenerator tool which transforms the safe behavioural model into a verification input of the NuSVM model checker. For that, we first derived the XML

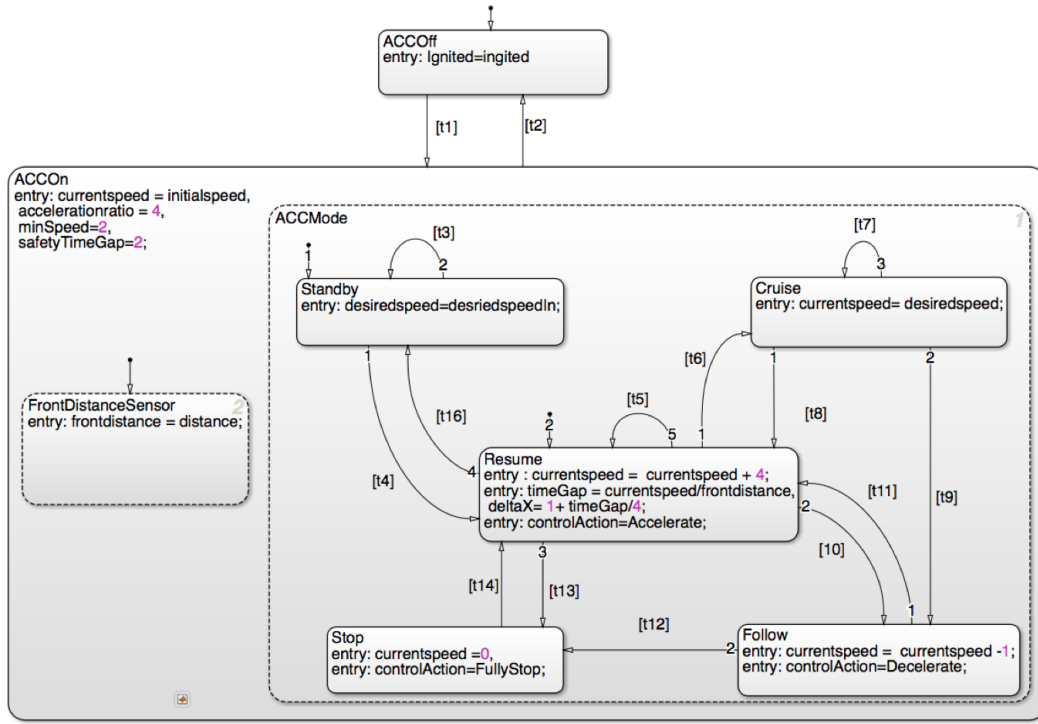


Fig. 19: The safe behavioural model of the ACC software controller

specifications of the Simulink's Stateflow model. Second, we took the XML specifications of both ACC simulator STPA file and the safe behavioural model as input to the STPA TCGenerator tool. The tool parses both files and generates the SMV model which maps all states, transitions and data variables, and LTL formulae of STPA software safety requirements of the safe behavioural model to SMV model specifications.

We updated the default values of each input data variable which are declared in the generated SMV model (e.g. *initial speed* (10.0), *desired speed* (45.0), *initial frontdistance* (150.0)). The value of current speed will be calculated by using equations 5. The value of time gap will also be calculated by using equation 2. The STPA TCGenerator tool runs the NuSMV 2.6.0 model checking tool to verify the generated SMV model file. The NuSMV model succeeded in verifying the generated SMV model within 0.29 seconds and no further errors were reported. NuSMV consumed 42.10 megabytes to store $2.31828e+17$ states and performed $2.97418e+09$ transitions. As a result, all LTL formulae were satisfied and there is no counterexample generated because the safe behavioural model itself was built from STPA software safety requirements.

6.3. Safety-based Test Case Generation from the Safe Test Model

After validating the correctness of the safe behavioural model, we used the STPA TCGenerator to generate a hierarchical tree of the safe behavioural model which shows the hierarchy levels of the safe test model. The STPA TCGenerator tool parses the tree of the safe behavioural model recursively by considering superstate decompositions *AND_STATE* (parallel) and *OR_STATE* (exclusive) to generate the safe test model as an extended finite state machine. As a result, the generated safe test model contains 7 states (after removing the superstates) and 32 transitions (after maintaining the

Table VIII: The safety-based test cases generated by STPA TCGenerator tool

| ID | Test Algorithm | Test Steps | Test Suite | Test Cases | Time (in Sec) | State Coverage | Transition Coverage | STPA SRR Coverage |
|----|----------------|------------|------------|------------|---------------|----------------|---------------------|-------------------|
| 1 | DFS | 10 | 1 | 119 | 3 | 6/7 = 85.7% | 23/32=71.9% | 32/32=100% |
| 2 | BFS | 10 | 4 | 24 | 1 | 6/7 = 85.7% | 17/32= 53.1% | 32/32=100% |
| 3 | Both | 10 | 5 | 249 | 2 | 7/7 = 100% | 18/32= 87.5% | 32/32=100% |

transitions of superstates). The tool automatically generates the traceability matrix between STPA software safety requirements and the safe behavioural model.

To generate the safety-based test cases from the safe test model of the ACC simulator, we first set the number of test steps to 10 and selected the three test coverage criteria (state, transition and STPA software safety requirements test coverage criteria) in the STPA TCGenerator tool. We selected the STPA software safety requirements coverage as the stop condition of the test case generating algorithm. We also set the test input value for each input data variable: *power* (true), desired speed (45 cm/sec), initial speed (10 cm/sec), front distance (150 cm). Finally, we ran the STPA TCGenerator tool three times to generate safety-based test cases from the test model, respectively: 1) depth-first search, 2) breadth-first search and 3) the combined algorithm. Table VIII shows the results of the generated safety-based test cases by each test algorithm. We could achieve 100% coverage of all the STPA software safe requirements which are linked to the safe test model in the traceability matrix. Figure 20 shows an example of the format of documenting each safety-based test case.

```

1  [Test Case ID] 2
2  [Test Suite ID] 2
3  [Related STPA SSRs]
4    RSSR1.1 , RSSR1.2 , RSSR1.3
5  [PreConditons]
6    desiredspeed=45.0
7    frontdistance=120.32
8    currentspeed=44.0
9    state=Resume
10 [Actions]
11   controlAction=Accelerate
12 [PostConditons]
13   currentSpeed=45.0
14   state=Cruise
15 [Comment]
```

Fig. 20: An example of a generated safety-based test case

Based on the traceability matrix between the model and the STPA software safety requirements, the *STPA TCGenerator* provides an *individual coverage* (how many test cases *TC* covered each *SSR*) by each test algorithm (shown in Fig. 21).

7. DISCUSSION

The idea behind the proposed approach is to integrate STPA safety analysis and its identification of the hazardous situations that the software can lead or contribute to semi-automatically with software testing. For this, we formalise the STPA software

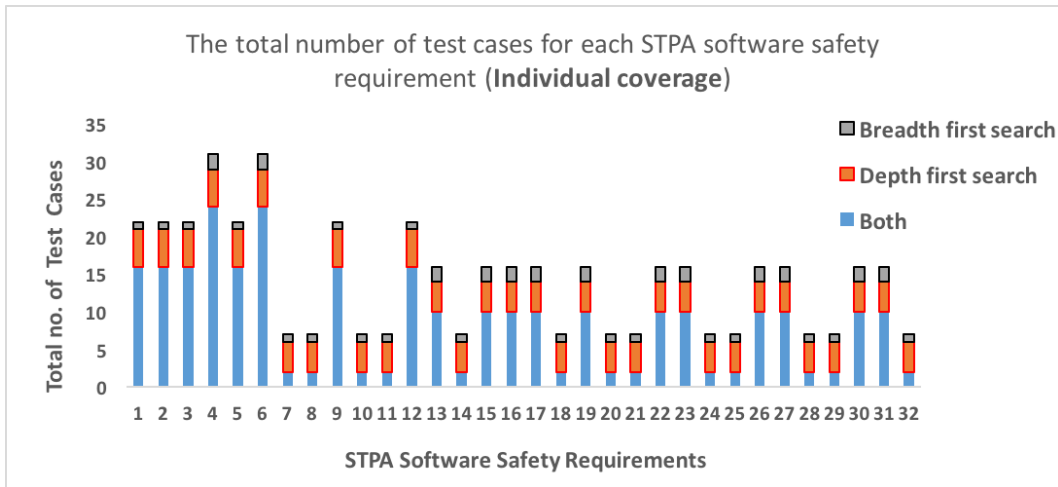


Fig. 21: The total number of test cases for each STPA software safety requirement

safety requirements into a formal specification and model the information derived from the STPA safety analysis into a test model. That helps us to focus the effort of testing by generating safety-based test cases for each software safety requirements. However, there are still some open issues and interesting challenges that require further research.

7.1. Visualisation of Process Model

The process model in the STPA control structure diagram is a very abstract model which shows only the safety-critical variables and states which have an effect on the safety of issuing the control actions by a software controller in the control structure diagram. It does not show how the software controller issues the control actions. Therefore, we use the statechart notation to visualise the relationships of the process model variables and describe the safe behavioural model. However, constructing a safe behavioural model from the STPA safety results by safety test engineers depends on the level of information which is available during the STPA safety analysis process (e.g. process model, and process model variables and values). Moreover, it is critical how this information describes the internal state of the software controller and the safety-critical software variables (e.g. interaction and environmental variables). Furthermore, visualising the safe behavioural model in a modelling tool such as Simulink requires user expertise in the modelling of dynamic behaviour to map the safety analysis specifications (process model, control actions and software safety constraints) into the Stateflow notation. Therefore, this point remains as future work to automatically provide a basic structure of the safe behavioural model from the process model information (e.g. states and its hierarchical levels) which is visualised in XSTAMPP. This will help the safety tester to understand the relationships between the critical system states, environmental and interaction variables which are documented in the process model of the software controller in the STPA control structure diagram.

7.2. The Correctness of the Safe Test Model

The manual construction of test cases is a hard, time-consuming and error-prone activity that requires deep knowledge and expertise. Furthermore, the manual building of a test model from system specifications with the purpose of generating test cases still

needs a proof of its correctness to ensure that the test model captures all specifications. A solution is to construct a test model for a given system and prove its correctness by transforming it into an intermediate model which is supported by a formal verification approach (e.g. model checker) to verify the generated model against its specifications. In addition, the specifications should also be mapped from informal text to the formal specifications. For this issue, we transformed the safe test model into the SMV model and verified it by using the NuSMV model checker to ensure that the safe test model satisfies the STPA specifications. However, the model transformation process also needs a proof of the correctness of the resultant model, even though the model checker did not induct any error. In our proposed approach, this issue remains as an open issue for future work.

7.3. Traceability Matrix

The automation of the test case generation process can lead to a large number of test cases that cover the same information. Reducing the number of generated test cases is a major factor in evaluating the effectiveness of an automated testing tool and the quality of the generated test cases. Therefore, we added a new test coverage criteria (STPA software safety requirements) to stop the test case generating algorithm when this criterion becomes 100% to ensure that each STPA safety requirement is covered at least in one test case. Furthermore, the first prototype of the *STPA TCGenerator* tool supports to generate test cases for each software safety requirement by automatically generating a traceability matrix by calculating the similarity degree of the matched tokens between the STPA software safety requirements and the safe test model. The traceability matrix contains all relevant transitions of each software safety requirement in the safe test model.

7.4. Process Model Variables Data Types

Another limitation is that the process model variables in the STPA control structure diagram visualised by XSTAMPP have no data types. Furthermore, XSTAMPP does not support multi-levels hierarchies of the process model of the software controller in the control structures. That makes ensuring and checking the consistency between the hierarchy levels of the process model in STPA and the Stateflow model in Simulink a big challenge. For example, the process model variable *ACC Active* in the *ACC* software controller has sub-process model variables such as control speed and *FrontDistance-sensor* which will be activated when the *ACC* state is active. Therefore, it requires human effort to define the process model hierarchy and map it to the Simulink Stateflow model hierarchy level.

8. CONCLUSION

In this paper, we introduced a systematic and semi-automatic approach to generate safety-based test cases based on the STPA safety analysis. Our approach concentrates on generating a set of test cases for each STPA software safety requirement. The generated test cases will be used to verify the safety of the software-intensive system under analysis. We also implemented an open-source tool support that automates the safety-based test cases generating approach. Furthermore, we illustrated the proposed approach with safety-critical software of an ACC system with stop-and-go function. The results show that deriving test cases based on the safety requirements is a practical and effective approach to generate different test cases to recognize software risks and assure the software quality.

As a future work, there are many interesting directions and trends to extend the research of safety-based testing for software-intensive systems and the automated tool support. We plan to improve the tool by considering the other Stateflow semantics

which were not addressed in our approach such as inner transitions and connective and history junctions. Furthermore, we aim to limit the number of the generated test cases, to improve the traceability matrix by adding information about the maximum number of test cases for each software safety requirement and also the priority value to generate a reasonable test case for each software safety requirement.

Furthermore, we plan to improve the process model in the control structure diagram by allowing the safety analyst to define the data type of each process model variable and draw the multi-hierarchy levels of the process model variables. Finally, we plan to evaluate the proposed approach and the tool support on a real software-intensive system with an industrial partner.

ACKNOWLEDGMENTS

The authors would like to thank Prof. Nancy Leveson, MIT, for her very careful review of our paper, and for the comments, corrections and suggestions that ensued.

We would also like to express gratitude to Lukas Balzer, University of Stuttgart, who worked with us to improve and build the XSTAMPP platform; Yannic Sowoidnich, University of Stuttgart, who developed XSTPA; Ting Luk-He, University of Stuttgart, who developed an Eclipse plugin for STPA TCGenerator; Rick Kuhn, National Institute of Standards and Technology, USA, who provides us the Automated Combinatorial Testing Tool (ACTS). We are grateful for their help, effort and time; and Kornelia Kuhle, University of Stuttgart, for her feedback on the text. We also would like to express our appreciation to the anonymous referees for their in-depth comments, suggestions and corrections to improve the quality of the paper.

REFERENCES

- Asim Abdulkhaleq and Stefan Wagner. 2015a. Integrated Safety Analysis Using Systems-Theoretic Process Analysis and Software Model Checking. In *Proc. 2015 SAFECOMP Computer Safety, Reliability, and Security* (2015).
- Asim Abdulkhaleq and Stefan Wagner. 2015b. XSSTAMPP: An eXtensible STAMP platform as tool support for safety engineering. In *2014 STAMP Conference, MIT*.
- Asim Abdulkhaleq, Stefan Wagner, and Nancy Leveson. 2015. A Comprehensive Safety Engineering Approach for Software-Intensive Systems Based on STPA. *Procedia Engineering* 128 (2015), 2 – 11. Proceedings of the 3rd European STAMP Workshop 5-6 October 2015, Amsterdam.
- D. Alberico, J. Bozarth, M. Brown, J. Gill, S. Mattern, and A. McKinlay VI. 1999. *Software System Safety Handbook: A Technical and Managerial Team Approach*. Joint Services Software Safety Committee.
- Rajeev Alur. 1999. Timed Automata. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV '99)*. Springer-Verlag, London, UK, UK, 8–22.
- Larry Apfelbaum and John Doyle. 1997. Model Based Testing. In *Software Quality Week Conference*. 296–300.
- Chonlawit Banphawatthanarak, Bruce H Krogh, and Ken Butts. 1999. Symbolic verification of executable control specifications. In *Computer Aided Control System Design, 1999. Proceedings of the 1999 IEEE International Symposium on*. IEEE, 581–586.
- C. Bourhfir, R. Dssouli, E. Aboulhamid, and N. Rico. 1997. *Automatic executable test case generation for extended finite state machine protocols*. Springer US, Boston, MA, 75–90.
- Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. 2005. *Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.
- Richard Castanet and Davy Rouillard. 2002. *Generate Certified Test Cases by Combining Theorem Proving and Reachability Analysis*. Testing of Communicating Systems XIV: Application to Internet Technologies and Services, Springer US, Boston, MA, 249–265.
- Roberto Cavada, Alessandro Cimatti, A. Charles Jochim, Gavin Keighren, Emanuele Olivetti, Marco Pistore, Marco Roveri, and Andrei Tchaltsev. 2010. NuSMV 2.6 User Manual. (jan 2010).
- Chunqing Chen and Jin Song Dong. 2006. Applying Timed Interval Calculus to Simulink Diagrams. In *Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM 2006, Macao, China, November 1-3, 2006, Proceedings*. 74–93.
- Chunqing Chen, Jun Sun, Yang Liu, Jin Song Dong, and Manchun Zheng. 2012. Formal modeling and validation of Stateflow diagrams. *STTT* 14, 6 (2012), 653–671.

- Kwang-Ting Cheng and A. S. Krishnakumar. 1993. Automatic Functional Test Generation Using The Extended Finite State Machine Model. In *Design Automation, 1993. 30th Conference on*. 86–91.
- Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. 2000. NUSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer* 2, 4 (2000), 410–425. DOI: <http://dx.doi.org/10.1007/s100090050046>
- Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. 1999. NUSMV: A New Symbolic Model Verifier. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV '99)*. Springer-Verlag, London, UK, UK, 495–499.
- S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. 1999. Model-based Testing in Practice. In *Proceedings of the 21st International Conference on Software Engineering (ICSE 99)*. ACM, New York, NY, USA, 285–294.
- Orlando Ferrante, Luca Benvenuti, Leonardo Mangeruca, Christos Sofronis, and Alberto Ferrari. 2012. *Parallel NuSMV: A NuSMV Extension for the Verification of Complex Embedded Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 409–416.
- Colin J. Fidge, Ian J. Hayes, A. P. Martin, and Axel Wabenhorst. 1998. A Set-Theoretic Model for Real-Time Specification and Reasoning. In *Proceedings of the Mathematics of Program Construction (MPC '98)*. Springer-Verlag, London, UK, UK, 188–206.
- Lloyd D. Fosdick and Leon J. Osterweil. 1976. Data Flow Analysis in Software Reliability. *ACM Comput. Surv.* 8, 3 (Sept. 1976), 305–330.
- A. Gill. 1962. *Introduction to the theory of finite-state machines*. McGraw-Hill.
- Youssef Hamadi, Said Jabbour, and Lakhdar Sais. 2008. ManySAT: a parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation* 6 (2008), 245–262.
- Grégoire Hamon. 2005. A Denotational Semantics for Stateflow. In *Proceedings of the 5th ACM International Conference on Embedded Software (EMSOFT '05)*. ACM, New York, NY, USA, 164–172.
- Grégoire Hamon and John Rushby. 2007a. An operational semantics for Stateflow. *International Journal on Software Tools for Technology Transfer* 9, 5 (2007), 447–456.
- Grégoire Hamon and John Rushby. 2007b. An operational semantics for Stateflow. *International Journal on Software Tools for Technology Transfer* 9, 5-6 (2007), 447–456.
- David Harel. 1987. Statecharts: a visual formalism for complex systems. *Science of Computer Programming* 8, 3 (1987), 231 – 274.
- David Harel and Amnon Naamad. 1996. The STATEMATE Semantics of Statecharts. *ACM Trans. Softw. Eng. Methodol.* 5, 4 (Oct. 1996), 293–333.
- Hyoung Seok Hong, Young Gon Kim, Sung Deok Cha, Doo Hwan Bae, and Hasan Ural. 2000. A test sequence selection method for statecharts. *Software Testing, Verification and Reliability* 10, 4 (2000), 203–227.
- SAE International. 1967. *Society for Automotive Engineers, Design Analysis Procedure for Failure Modes, Effects and Criticality Analysis (FMECA)*, ARP926. Warrendale, USA.
- JPL. 2000. Report of the Loss of the Mars Polar Lander and Deep Space 2 Missions. (2000).
- Y. G. Kim, H. S. Hong, D. H. Bae, and S. D. Cha. 1999. Test cases generation from UML state diagrams. *IEEE Proceedings - Software* 146, 4 (Aug 1999), 187–192.
- J. Kloos, T. Hussain, and R. Eschbach. 2011. Risk-Based Testing of Safety-Critical Embedded Systems Driven by Fault Tree Analysis. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*. 26–33.
- M. K. Kuan. 1962. Graphic programming using odd or even points. (1962).
- D. Richard Kuhn, Raghu N. Kacker, and Yu Lei. 2013. *Introduction to Combinatorial Testing* (1st ed.). Chapman & Hall/CRC.
- Nancy Leveson. 2000. Completeness in Formal Specification Language Design for Process-control Systems. In *Proceedings of the Third Workshop on Formal Methods in Software Practice (FMSP '00)*. ACM, New York, NY, USA, 75–87.
- N.G. Leveson. 2011. *Engineering a Safer World: Systems Thinking Applied to Safety*. MIT Press.
- M. Li and R. Kumar. 2012. Model-based automatic test generation for Simulink/Stateflow using extended finite automaton. In *2012 IEEE International Conference on Automation Science and Engineering (CASE)*. 857–862.
- Robyn R. Lutz. 2000. Software Engineering for Safety: A Roadmap. In *Proceedings of the Conference on The Future of Software Engineering (ICSE '00)*. ACM, New York, NY, USA, 213–226.
- MathWorks. 2016. The MathWorks, Inc. Simulink, 2015. Version R2015b. (Jan 2016).
- Kenneth L. McMillan. 1993. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA.

- G. H. Mealy. 1955. A method for synthesizing sequential circuits. *The Bell System Technical Journal* 34, 5 (Sept 1955), 1045–1079. DOI: <http://dx.doi.org/10.1002/j.1538-7305.1955.tb03788.x>
- B Meenakshi, Abhishek Bhatnagar, and Sudeepa Roy. 2006. Tool for translating simulink models into input language of a model checker. In *International Conference on Formal Engineering Methods*. Springer, 606–620.
- Minister of Defence. 1991. Hazard Analysis and Safety Classification of the Computer and Programmable Electronic System Elements of Defence Equipment, Interim Defence Standard 00-56, Issue 1. (1991).
- NASA-GB- 8719.13. 2004. *NASA Software Safety Guidebook*. NASA.
- A. Jefferson Offutt, Shaoying Liu, and Aynur Abdurazik. 2003. Generating Test Data From State-based Specifications. (2003).
- C. S. Pasareanu, J. Schumann, P. Mehltz, M. Lowry, G. Karsai, H. Nine, and S. Neema. 2009. Model Based Analysis and Test Generation for Flight Software. In *Space Mission Challenges for Information Technology, 2009. SMC-IT 2009. Third IEEE International Conference on*. 83–90.
- Amir Pnueli. 1977. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (SFCS '77)*. IEEE Computer Society, Washington, DC, USA, 46–57.
- Alexander Pretschner. 2001. Classical Search Strategies for Test Case Generation with Constraint Logic Programming. In *In Proc. Formal Approaches to Testing of Software*. BRICS, 47–60.
- S. J. Prowell. 2003. JUMBL: a tool for model-based statistical testing. In *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*. 9 pp.–.
- Felix Redmill. 2004. Exploring Risk-based Testing and Its Implications: Research Articles. *Softw. Test. Verif. Reliab.* 14, 1 (March 2004), 3–15.
- Jun Sun, Yang Liu, Jin Song Dong, and Chunqing Chen. 2009a. Integrating specification and programs for system modeling and verification. In *Theoretical Aspects of Software Engineering, 2009. TASE 2009. Third IEEE International Symposium on*. IEEE, 127–135.
- Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. 2009b. PAT: Towards flexible verification under fairness. In *International Conference on Computer Aided Verification*. Springer, 709–714.
- John Thomas. 2013a. *Extending and automating a systems-theoretic hazard analysis for requirements generation and analysis*. Ph.D. Dissertation. MIT.
- John Thomas. April 2013b. *Extending and automating a systems-theoretic hazard analysis for requirements generation and analysis*. dissertation. MIT.
- J. Thomas, F. Lemos, and N. Leveson. November, 2012. Evaluating the Safety of Digital Instrumentation and Control Systems in Nuclear Power Plants. *MIT Technical Report* (November, 2012).
- J. Thomas and G. N. Leveson. 2011. Performing Hazard Analysis on Complex, Software- and Human-Intensive Systems. In *29th International System Safety Conference* (2011).
- UML. 2004. *Unified Modeling Language Specification* (version 2.0 ed.). Object Management Group (OMG).
- Hasan Ural. 1987. Test sequence selection based on static data flow analysis. *Computer Communications* 10, 5 (1987), 234 – 242.
- Mark Utting and Bruno Legeard. 2007. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
- P. Venhovens and Adiprasito B. Naab, K. 2000. Stop and go cruise control. *Seoul 2000 FISITA World Automotive Congress 2000* (jun 2000), 396.
- W. Vesely, F.F. Goldberg, N.H. Roberts, and D.F. Haasl. 1981. *Fault Tree Handbook NUREG-0492*. U.S. Nuclear Regulatory Agency, Washington.
- A. Windisch. 2009. Search-based testing of complex simulink models containing stateflow diagrams. In *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*. 395–398.
- Y. Yang, Y. Jiang, M. Gu, and J. Sun. 2016. Verifying Simulink Stateflow model: Timed automata approach. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 852–857.
- Yuan Zhan and John A. Clark. 2008. A Search-based Framework for Automatic Testing of MATLAB/Simulink Models. *J. Syst. Softw.* 81, 2 (Feb. 2008), 262–285.
- F. Zimmermann, R. Eschbach, J. Kloss, and T. Bauer. 2009. Risk-based Statistical Testing: A refinement-based approach to the reliability analysis of safety-critical systems. *Proceedings of the Spring TAV Workshop* (2009).